Coordinating multiset transformers

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam,
op gezag van de Rector Magnificus prof. dr J.J M. Franse
ten overstaan van een door het college van dekanen ingestelde
commissie in het openbaar te verdedigen in de Aula der Universiteit
op maandag, 27 oktober 1997, te 13:00 uur door
Hugh Edmund McEvoy
geboren te Leeds, Engeland

Promotores:	prof.	dr. L. O. Hertzberger
	prof.	dr. C. Hankin
Commissie:	prof.	dr. H. Kuchen
	prof.	dr. H. Sips
		dr. W. Vree

Faculteit der Wiskunde, Informatica, Natuurkunde en Sterenkunde
Kruislaan 403,
1098 SJ Amsterdam
The Netherlands

ISBN: 90-74795-76-5

The front cover shows a Goblin program to generate Dolly, the world's first autocatalytic sheep. The back cover shows the result of executing the program on the singleton multiset containing Dolly.

Contents

1 Introduction: coordination is everywhere **7**
- 1.0.1 Coordination using control-flow . . . 9
- 1.0.2 Coordination through interoperability . . . 10
- 1.0.3 Coordination using multiset transformation . . . 11

1.1 Introduction to Γ . . . 12
- 1.1.1 The syntax and semantics of Γ . . . 13
- 1.1.2 Examples of Γ programs . . . 14

1.2 The friends of MST . . . 17
- 1.2.1 Parallel and distributed computing . . . 17
- 1.2.2 Specification formalisms . . . 19
- 1.2.3 Process algebrae and calculi . . . 21
- 1.2.4 Chemical abstract machines . . . 25
- 1.2.5 Language paradigms . . . 25
- 1.2.6 Formal languages . . . 27
- 1.2.7 Physical modelling . . . 27

1.3 Open problems in MST . . . 31
- 1.3.1 Efficiency . . . 32
- 1.3.2 Compositionality . . . 32
- 1.3.3 Flexibility . . . 33
- 1.3.4 Γ's problems . . . 34

1.4 The motivation for this thesis . . . 35
- 1.4.1 The structure of this thesis . . . 36

2 A unified semantic framework for parallel transformation **37**

2.1 Introduction . . . 38

2.2 Introducing PT . . . 39
- 2.2.1 The syntax of PT . . . 40
- 2.2.2 Anatomies of a selection function and interest tuple . . . 41
- 2.2.3 The formal semantics of PT . . . 43

2.3 A posse of parallel languages . . . 44
- 2.3.1 Γ . . . 46
- 2.3.2 Γ^{μ}: Γ with loops and single-shot functions . . . 46
- 2.3.3 Calculus of Gamma Programs (CGP) . . . 47
- 2.3.4 A context-sensitive Gamma (CSΓ) . . . 49
- 2.3.5 A context-insensitive Gamma (CIΓ) . . . 49
- 2.3.6 State sets . . . 49

2.3.7 Non-linearity 50
2.3.8 AlChemy: chemical reactions modelled using λ-terms 50
2.3.9 Abortion and deadlock 50
2.4 Context sensitivity and interleaving choice 51
2.5 Synchronous and non-synchronous reductions 54
2.5.1 Some results linking Γ, Γ^{μ}, CGP and CGP^{μ} 55
2.6 A taxonomy of Γ-like languages 57
2.7 Fairness 57
2.8 Future work 58
2.9 Conclusions 59

3 State synchronisation and data contexts 61
3.1 Introduction 61
3.2 Data contexts in PT 62
3.3 Synchronous state update in PT 64
3.4 Encoding parallel operators 68
3.4.1 A non-prescriptive parallel operator 68
3.4.2 A prescriptive parallel operator 68
3.5 Combining state synchronisation and data contexts 70
3.6 Encoding Γ in PT with $\mathcal{S}_{G1}$ and $\mathcal{S}_{G2}$. 77
3.7 Addendum: the MSTL Goblin 78
3.8 Future work 82
3.9 Conclusions 83

4 Modelling stochastic phenomena using MST 85
4.1 Models of stochastic phenomena 85
4.1.1 The structure of this chapter 86
4.2 Stochastic versus non-deterministic computation 87
4.3 Stochastic MST program reduction 88
4.3.1 Random number generators 88
4.3.2 Functions chosen with probabilities determined statically 89
4.3.3 Functions chosen with probabilities determined dynamically 89
4.3.4 Subset selection with probabilities determined statically 90
4.3.5 Dynamic variation of probabilities for subset selection 91
4.3.6 Termination with a statically-determined probability . 92
4.3.7 Termination with a dynamically-determined probability 92
4.4 Example stochastic applications 93
4.4.1 Monte Carlo integral approximation 93
4.4.2 Diffusion-limited aggregation 94
4.4.3 A genetic algorithm: an example using normalisation . 95
4.4.4 Simulated annealing 96
4.5 Future work 99
4.6 Conclusions 99

5 Modelling environmentally-sensitive growth using MST 101
5.1 Introduction . . . 101
5.2 Modelling Growth of Ramifying Objects . . . 103
5.2.1 Collisions with a fixed object . . . 105
5.2.2 Growth towards the light . . . 107
5.2.3 Non-self intersection . . . 109
5.2.4 Combining non-self intersection with photo-sensitivity 111
5.2.5 A Cellular Automaton . . . 111
5.3 Conclusions . . . 112

6 Conclusions 115
6.1 Summary of the chapters . . . 116
6.2 Future work . . . 119
6.2.1 Theoretical tasks ahead . . . 119
6.2.2 Practical tasks ahead . . . 120
6.2.3 The multiset: there is no silver billet . . . 121
6.3 Exeunt . . . 121

7 Samenvattingen/summaries 123
7.1 Nederlands . . . 123
7.2 English . . . 125

A Proofs 129
A.1 Γ to PT proof . . . 129
A.1.1 Simple programs . . . 129
A.1.2 Non-simple programs . . . 131
A.2 CGP to PT proof . . . 133
A.2.1 Simple programs . . . 133
A.2.2 Non-simple programs . . . 135
A.3 Γ to PT proof . . . 137
A.3.1 Simple programs . . . 137
A.3.2 Non-simple programs . . . 140

Acknowledgments

This is the bit where I can ramble on for pages about all of the people who have helped me in any way during my years in Amsterdam.

First and most of all, I would like to thank Pieter Hartel for all that he has done for me. Not only did he give me a lot of support and help in all aspects of life when I got here (such as helping me open a bank account: "The Postbank is a good one, Hugh. It doesn't cost much to keep your money there"!!! And this to an Englishman who was used to being *paid* to put his money in the bank), but he also gave me a lot of time in the initial parts of my research. Since he moved to Southampton, our meetings have been less frequent but no less friendly. Above and beyond the call of duty, he let me 'anti-squat' in his house when the oh-so-marvellous Dutch housing system demonstrated that it neither works for foreigners (where 'foreigner' is defined as somebody who doesn't come from the district of Amsterdam) nor is interested in the troubles of those for whom it doesn't work. So much for free movement of labour within the EU...

My work was supervised by a posse of people at the University of Amsterdam, most of whom had different ideas about what I ought to do. Thus is intradisciplinary work defined. The original plan was to get two groups (declarative systems and scientific computing) to collaborate. Whether or not the collaboration succeeded, we leave to the reader to judge. In any case, offspring of this marriage haunt the pages of this thesis. For all their help and patience with my disinclination to listen to anybody, I would like to thank Bob Hertzberger, Wim Vree and Peter Sloot. As with all coordination enterprises, some communication protocols were necessary. If I have irritated any of you with my lack of appreciation for the finer points of diplomacy, I am sorry.

Chris Hankin has my warmest thanks for being the chap who started me on this road back in 1993, when he suggested that I do my Master's project on an implementation of Gamma. I'm not sure that he knew what he was getting himself into. Having been there at the start of the journey, I am honoured that he is still travelling with me. And I still owe him a beer. Better make it two.

During November and December of 1996, I travelled to the University of Calgary, Canada. There, I worked with some truly wonderful people and learned lots of fascinating things. Some of those things I even learned at work! My warmest gratitude goes to Przemek Prusinkiewicz for his sup-

portive collaboration, his friendship and the music that we played together. I shall never forget it. In the same breath, my thanks go out to all my (temporary) colleagues at the U of C: you know who you are! A special mention must go to Camille Sinanan for putting up with me, Eric Galin, Brian 'toothbrush' Wyvill and Robin Cockett. Maybe you haven't seen the last of me. Peter Sloot also deserves a large round of applause: he was the catalyst for the whole adventure and, being a catalyst, was not consumed by the reaction which followed. Just as well, really.

Marcel Beemster deserves a doughnut (which *is* the same as an oliebol) for all of his help, his friendship and his belaying. I can't imagine a better person with whom to share an office (but is it mutual?). For his friendship, very definite opinions and Magic campaigns, Jon Mountjoy must be thanked, as must Nikki Mountjoy for the first of these.

Over the last years, I have asked many people many questions about many subjects. I hope that not too many of them (the questions, that is!) were stupid, but in any case, thanks go to Chris Verhoef, Andy Pimentel, Theodossis Papathanassiadis (hope I spelled that right), Toto van Inge, Anne Troelstra, Krzysztof Apt, Jeroen Voogd, Arjen Schoneveld, Benno Overeinder, Alfons Hoekstra and a cast of thousands for trying to answer them. If I have forgotten anyone, it was intentional. In the same breath, thanks go to Jaap Kaandorp for having a very great influence on my work. It all started with me asking him about his thesis, which I saw on somebody's bookshelf. I found and continue to find his work on modelling sponges, corals and other fish ;-) fascinating. His desk is a mess, though.

On the social side, I'd like to acknowledge the friendships I've made in this beautiful city: Andrea Brands, Kasper "Dook" van Benten, Martijn van Puffelen, Marion Kolader and, of course, The Band! You know who you are. Thanks to you all for the pinball wars and syncopation (even if the latter *was* unintentional).

Finally (unless I think of someone else), I'd like to thank D minor for being the saddest of all keys.

Hugh McEvoy, Amsterdam. July 1997

"I should have liked to produce a good book, This has not come about, but the time is past in which I could improve it."

Ludwig Wittgenstein, Preface to 'Philosophical Investigations', Blackwell, England, 1958

For music and for Heidi, the loves of my life.

Forward

The myriad of computer applications in the mass-market, such as Quake[1] and WordPerfect[2], makes it easy to get blinded by the ease with which computers can accomplish the most complex tasks. In particular, the widespread use of parallel and, more commonly, distributed, computing environments within the business community may perhaps lead one to think that all the problems of computing are solved. Alas, this is not so. What is often less obvious to the casual user of software, is that controlling the complexities of software products' interaction with themselves and with other systems in a computer still owes more to necromancy than to science. The ability to coordinate a number of components and mediate their interactions with each other are only now beginning to be studied in isolation from particular engineering solutions.

This thesis concerns itself with an examination of mechanisms for controlling software scheduling and intercommunication. These two areas rally loosely under the banner *coordination*. Our particular viewpoint is essentially abstract, with the state of the machine being represented by an unstructured multiset (bag) of information, which is manipulated by programs. Such a computation model is termed a multiset transformation language (MSTL). We can view a sequence of successive (possible) contents of the multiset as a consequence of the programs' interactions with that multiset. The programs are themselves scheduled by a higher-level mechanism, a formally-defined language which is able to schedule tasks sequentially, conditionally or in an interleaved fashion. We hereby obtain a powerful tool for understanding function coordination in a quite general setting[3]. Regrettably, our optic also clouds our vision somewhat, in particular because our multiset makes irritatingly cumbersome investigation of systems requiring structured state. However, our undertaking is far from hopeless, as witnessed by our ability to give, in terms of our formal framework, a reasonably systematic analysis of interleaving operators and a number of not entirely trivial applications. These are our cause for cheer.

The investigative orientation taken in this thesis is essentially *intra-*

[1] A marvellous game from iD software in which players massacre horrible monsters and occasionally each other.

[2] WordPerfect is a trademark of the WordPerfect corporation. WordPerfect is much less violent than Quake.

[3] But not completely general: our simplest functions are atomic which, in practice, may only be the case at the processor level.

disciplinary (which is not nearly so fashionable as *inter*-disciplinary research). Steering the investigations is the firm conviction that, without keeping a model's applications firmly before the mind's eye, developing that model is unlikely to yield significant benefits. This peril is, in this author's opinion, particularly insidious with theoretical work, when assumptions made to ease formal investigation sometimes render a model inapplicable in practice. We shall encounter examples of this along our path. At the same time, approaching an abstract view of coordinating systems as one would a programming project is similarly inadvisable and, if followed by the majority, likely to result in long-term impoverishment of computer *science* (if it is, indeed, a science). My hybrid orientation explains the rather eclectic collection of investigations here reported. Picking up the ball and running with it as far as possible might be considered by some the only way to play the game, but a wider perspective seems to me to be at least as important.

This thesis contains two main tributaries. Firstly, textually speaking, come the investigations of the formal nature of multiset transformation languages. Interleaving composition of programs is singled out for particular consideration involving, as it does, rather complex relationships with logical parallelism. Some of what I say is distinctly heterodox: shortcomings of certain well-known models *qua* interleaving and parallelism, are claimed. These shortcomings come to light as one tries to write non-trivial programs. After initially believing interleaving to be a relatively straightforward issue, I gradually came to the conclusion that interleaving is actually a Pandora's box of hornets' nests (yes, it's that dangerous).

The second tributary is the applications of MSTLs themselves, which hail mainly from the scientific modelling community. In these chapters, my aim is twofold: firstly, to drive multiset transformation development with applications from the 'real world' and, secondly, to show that multiset transformation models actually offer something of interest to the physical modelling community.

The fractal border between the theoretical and the practical is hard to discern, in part because I try to make clear where the applications have motivated the theory throughout the thesis. The result of this is that I jump backwards and forwards between the two sub-disciplines. I do not apologise for this. I believe that the benefits to an understanding of multiset transformation yielded by the interplay between the theoretical and practical are more numerous and more important than the benefits yielded by either of the tributaries in isolation.

Chapter 1

Introduction: coordination is everywhere

Wherever we look, we find the interaction between agents being controlled and mediated by other agents. Businesses schedule their projects, assigning starting dates and resources to each. Employers coordinate their employee's actions and interactions. Employers also coordinate with other businesses in providing products for each other and responding to each others competitive pressures. Committees coordinate with each other to reach agreement and even coordinate with each other to agree on what constitutes agreement [137]. Swimming fish coordinate their actions so that they always stay within a certain distance of their shoal-mates [73, 56]. A window manager coordinates the interaction between the applications and the screen, handling requests for windows to be iconified or enlarged and passing input and output between the user, the screen and the application programs [68]. Imperative programming languages control the ordering of expression evaluations and function calls through the language's control-flow constructs. Conditionals form a bridge between data- and control-flow, executing a certain sequence of actions depending on the values of the data. The list of coordinating mechanisms goes on and on.

Despite the obvious appeal of being able to say something coherent about coordination in general, this thesis concerns itself only with those aspects of coordination related to computer software. With this caveat, we can view coordination as enabling interaction. Informally, this can occur in two ways.

1. By making decisions about the order in which coordinated programs should be executed. The control mechanism executes (parts of) the component programs in the desired order. Control-flow constructs examined in this thesis include sequential execution, conditional execution, non-deterministic choice and interleaved execution. Others may also be possible. This form of coordination we term *control-flow* coordination.

2. By empowering components to communicate with each other. For all forms of inter-process communication, it is vital that the components

be able to communicate with each other in principle. This kind of coordination we term *coordination through interoperability*. Interoperability can be provided in a number of ways, each of which imposes certain structure on the communication. Some examples are given below. Whether interoperability empowers communication of data or communication as synchronisation [69] is not considered relevant in this thesis. The important question is: what is required before communication can take place at all?

These two aspects of coordination are adequate to draw a line in the sand between considering coordination as prescriptive undertaking—using control-flow to determine the order in which programs can execute—and as a non-prescriptive undertaking—allowing the data-flow imposed by and on the component programs to determine the order in which they can execute.

Despite the two rather different uses of 'coordination', they are certainly not unrelated. For example, Linda [59, 58] and the Common Object Request Broker Architecture (CORBA) [132] only provide interoperability for software components. These two models impose no explicit control-flow on the programs. Instead, the order of program executions is determined only by the ordering of message receiving and sending between processes. Both languages are added to a conventional base language, whose control-flow constructs determine the order in which messages are sent between processes. A different approach can be seen in Γ [17, 16] and CGP [37], in which both interoperability and control-flow coordination mechanisms are manifested. In such languages, the order of program execution can be made explicit using control-flow operators within the coordination language (see Section 1.1 for details of Γ's control-flow operators). In addition to explicit control-flow, further orderings on program executions can be enforced through programs' data-flow characteristics, as in Linda and CORBA. Finally, the language Facility [1] adds control-flow style coordination to CORBA-like languages. More details of all these coordination languages will be given in due course. The distinction between control-flow coordination and coordination by interoperability is rarely stressed, perhaps because work on one often implicitly assumes the presence of the other.

In recent years, coordination has arisen as a discipline in its own right [1]. Within the scientific coordination community, models are proposed to handle one or both aspects of coordination, although the general trend seems to be study of control-flow coordination. At the same time, an explosion of interest in interoperability in the business community has led to development of standards such as CORBA [132] as a way of integrating systems written at different times, by different people, for different platforms, in different languages.

The rest of this introduction is structured as follows. We first discuss the two forms of coordination—control-flow and interoperability—in greater detail. Next, we introduce MST, together with an example MSTL called Gamma (Γ) [20, 65] which was proposed by Banâtre and Le Métayer. We give a few short examples of Γ programs to whet the readers' appetite for more. We then illustrate the similarities between languages such as Γ and

other programming models which have arisen at different times in different disciplines. The prevalence of MST-like ideas in a variety of fields suggests that MST has something to offer. Unfortunately, we have found Le Métayer's paradise only to discover that our abiding there is on sufferance: extant MSTLs suffer from a number of difficulties, which are discussed in Section 1.3. In Section 1.4, we give a brief overview of the work which has been undertaken in this thesis. The work consists of theoretic attempts to overcome or, at least, ameliorate some of the problems of current MSTLs, coupled with attempts to generalise the model and to understand how classes of applications can be encoded within the (extended) model.

1.0.1 Coordination using control-flow

Imperative programming languages contain constructs which dictate the control flow of a program. Examples are many and include sequential composition, parallel composition, choice, conditionals and gotos. Control-flow constructs can be thought of as coordination constructs, as they control the order in which program fragments are applied to the state. Similarly, they can be thought of as communication constructs, where the information communicated only concerns the execution status of a program fragment. An illustration is given in Figure 1.1. Information can only be communicated from the left- to the right-hand side of a sequential composition: no communication in the opposite direction is possible. As we shall see in Chapter 2, interleaving compositions can also be viewed as communication enablers between the composed program fragments. In that chapter, we see a good example of this property: in Γ [65], interleaved program fragments can only execute if their possible reductions are at least as 'interesting' as those offered by other programs in the composition. Communication can be viewed as bidirectional, with composed programs deciding between them which can execute.

Control-flow constructs are also communication *limiters*, in the sense that they impose *a priori* restrictions on the ways in which processes can communicate. A good example of this is a variable which is changed by a program A and then read by a program B. If A is to the right of B in a sequential composition, such communication cannot take place.

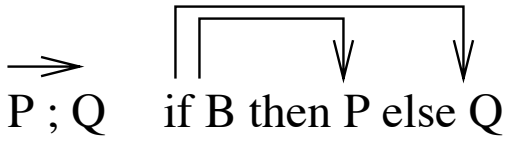


Figure 1.1: Control as communication. We can view control constructs in an imperative language as specifying (very limited) communication between program fragments. The arrows indicate the direction of the communication.

Where, exactly, communication ends and control-flow begins is therefore

not entirely clear. However, what *is* clear is that imposing control flow on a set of tasks which would otherwise execute in parallel radically changes the communication possibilities between the processes. If all programs are executing in an interleaving/parallel composition, coordination occurs only in the sense in which data-flow determines the order in which programs are executed.

1.0.2 Coordination through interoperability

Increased performance, together with decreasing hardware costs, has made it possible for software vendors to write larger and more complex programs. In part, this is merely undesirable—too many organisations produce buggy code and too many produce code which is not optimised for space or time usage, even in those cases when optimisation is desirable[1]. Both these problems are exacerbated by increased software complexity. However, the increased use of computers in workplaces and homes requires that they become more friendly to the user: offering better interfaces, more help and more functionality. Furthermore, computers are becoming so much a part of the fabric of the modern world that they are being used for things which Alan Turing probably never seriously considered[2]. The spreading use of computers from the laboratory to the home, necessitates vastly larger and more complex software. With larger software systems comes the need to understand how such systems can be designed, built and maintained; not as one-off research projects but as software products that can be modified or augmented to meet each new challenge.

Much of large systems' complexity arises from the plethora of parts which must be built and which must cooperate in solving a problem. Common sense and thousands of person-years of design experience [78] suggest that we have to implement large systems in small parts which work together. Furthermore, we must be able to replace parts of our code with other parts with similar functionality, add completely new parts to already working systems and use old parts again in new systems. All this has to be achieved with minimum difficulty. In open systems [41], the problems are exacerbated by the need to add new functionality and remove old functionality at runtime.

As an attempt to mitigate the problems of building large, open, distributed systems, the Object Management Group (OMG) has proposed the Common Object Request Broker Architecture (CORBA) [132] as a standard for interoperability between objects written by different vendors in different languages. CORBA allows the definition of interfaces between objects, with the interfaces defined in the Interface Definition Language (IDL). Objects wishing to use CORBA must have their interfaces written in IDL so that a proprietary compiler can generate executables using the IDL specification and the programmer's source (which can be written in any of a number of languages). The result is an executable which can invoke methods statically or dynamically in any other CORBA object on the network. Routing of

[1] Such as for an operating system.
[2] e.g. Quake.

messages is performed by the Object Request Broker (ORB)[3]. If an object is re-distributed to another node on the network, the ORB automatically tracks it and routes messages to it. This rôle of the ORB eases application development considerably: the application programmer merely has to decide which message must be sent to which object. The ORB then locates that object, which can be anywhere on the network, and ensures that the message is delivered.

Clearly, the form of coordination enabled by CORBA is that of interoperability: no control constructs schedule the objects. Instead, the ordering on the objects' own method invocation schedules the ordering of execution. This ordering is not imposed (or even available) to the ORB being, as it is, a part of the programmer's code outside the IDL code.

1.0.3 Coordination using multiset transformation

This thesis discusses coordination from the perspective of computations over an unstructured state (a multiset). A multiset is the least constrained possible communication medium. The agents are programs or processes, while communication is mediated by data which are placed in the multiset. The available control constructs are constrained as little as possible and the application areas are left as open as possible. This, very general, model is known as *multiset transformation* (MST) and its associated languages as *multiset transformation languages* (MSTLs). Similarly, a program written in an MSTL we christen a *multiset transformer*. We shall see many examples of such programs in the pages which follow. This thesis therefore interests itself in coordination of multiset transformers; hence its title.

The concept of programming by multiset transformation has been attributed to Jean-Pierre Banâtre and Daniel Le Métayer [16, 19, 18]. Their idea is compellingly simple and consists of two parts:

1. The state is a multiset. A multiset differs from a set in that the former allows multiple instances of the same value to exist as distinct entities, unlike with a set. A multiset is completely unstructured, so our state is effectively a virtual shared state. Data placed in the multiset by one agent can be removed by other agents (or even by the same agent at a later time). Those familiar with Linda [58] will recognise a tuple space being used for generative communication. We shall see numerous examples of the flexibility a multiset offers in the chapters ahead. The multiset's lack of structure also renders access to the multiset non-deterministically: if several elements in the set satisfy a predicate, then the choice of which one will be removed is made non-deterministic. The interaction between programs and state is specified formally.

2. Computation takes place according to a 'chemical-reaction' metaphor. The idea is that multiset elements are thought of as 'molecules' of data. An analogue of Brownian motion brings molecules together. When the

[3]As usual in industrially-sponsored activities, TLAs (Three-Letter Abbreviations) abound.

> molecules collide, they 'react' with each other if they satisfy certain conditions, which are specified in the program. If a reaction takes place, then the 'reagents' are replaced by their products. Eventually (assuming that the program terminates), all of the reactions which can take place have done so, and the multiset is then unable to change i.e. is 'stable' with respect to the given program. The final multiset is the outcome of the computation.

For the sake of this thesis, we define a multiset transformation language as a language possessing either one of these properties. For each language we describe, we explain which of two these properties it possesses.

What is the added value of this programming model? In their original papers [17, 16, 19], Banâtre and Le Métayer claimed that their motivation was to provide a high-level parallel programming language that could be used to provide a demonstrably correct initial version of a program. This version would exhibit a minimum of *extra-logical sequentiality*. In other words, the program so expressed would be as parallel as possible, given the logic of the algorithm. Using this initial, highly-parallel, program as a starting point, an executable program could be developed. This executable could be either derived from the original program by a process of refinement or compiled directly from the initial program.

More recently, there have been proposals to use Γ as a coordination language [1] for heterogeneous software components. The intention is to able to apply control-flow coordination to a number of software components, whose data would be communicated via a tuple space (the multiset). In this way, both the control-flow and interoperability styles of coordination would be ensured. One of the most interesting properties of multisets as communication media, noticed in the early Linda work [59], is that messages placed in the the multiset can be retrieved by processes started after termination of the message's sender. This form of communication with the unborn was christened *generative communication* [58], indicating the offered possibility of communicating with future generations. Furthermore, an unstructured multiset can be seen as an abstraction from a communication medium: an abstraction which constrains as little as possible the choice of medium used in a real system.

1.1 Introduction to Γ

Of all multiset transformation languages, the one with which this thesis is most concerned is Γ [65, 20]. We choose Γ because it is one of the best-known multiset transformation languages and has a simple and well-defined formal semantics. Our starting with Γ colours a lot of what is said in this thesis, so we here present a formal and informal introduction to Γ to give the reader an appreciation of the language.

$$
\begin{array}{lcll}
p \in P & \stackrel{\text{def}}{=} & (B, A) & \text{(primitive function)} \\
 & & (P) & \text{(brackets)} \\
 & & P_1 + P_2 & \text{(interleaving composition)} \\
 & & P_2 \circ P_1 & \text{(sequential composition)}
\end{array}
$$

$$
\begin{array}{lll}
\textit{where}: & & \\
P & = & \text{program} \\
B & = & \text{boolean conditions} \\
A & = & \text{action functions} \\
M & = & \text{multiset}
\end{array}
$$

Figure 1.2: The abstract syntax of Gamma. The types of all functions are omitted here: they can be seen in Chapter 2. The rôle of the multiset will become apparent in due course.

1.1.1 The syntax and semantics of Γ

In the Γ model, an initial multiset is repeatedly transformed as functions are applied to the elements of the multiset. The results of function applications are placed in the multiset and are then re-used as function arguments until no more applications are possible. This occurs either when the set contains too few elements to provide enough arguments to any function, or when no permutation of the elements satisfies the boolean condition of any function. A number of Γ programs have appeared in the literature, for example sorting programs [65], graph problems [16], string processing programs [19] and edge-detection programs [40]. The abstract syntax of Γ is shown in Figure 1.2 and its structural operational semantics (SOS [114]) in 1.3. The SOS of Γ is described in terms of a one-step transition relation on ⟨ program, multiset ⟩ pairs (*configurations*). A one-step transition from configuration to configuration we term a *reduction*. A *reduction path* is a sequence of reductions, normally terminating with a configuration whose program component is empty (i.e. an empty program, multiset pair).

The basic unit of interaction between the multiset and the program is the primitive function. A primitive function is applied to a vector consisting of zero or more elements $\vec{m}$ of the multiset. In a slight abuse of notation, we write $\exists \vec{\sigma} \subseteq \sigma$ for 'there exists a subset of elements of the multiset σ which can be ordered as a vector $\vec{\sigma}$'. If the boolean condition (the B in (B, A)) is satisfied (Rule $\mathbf{App}_1$), then the elements $\vec{m}$ are replaced by $A\vec{m}$. We say that the multiset has been *rewritten*. Notice that primitive functions are recursive: if a primitive function is successfully applied to the multiset, then that function 'regenerates' itself and is again available. If the boolean condition on the elements is *not* satisfied (Rule $\mathbf{App}_2$), then the primitive function is reduced to the empty program and the multiset is left unchanged. We construct programs from programs using two program composition operators (primitive functions are also programs). For each, we describe its behaviour in intuitive terms. The first is written '∘' and is a right-associative sequential

$$\frac{\exists \vec{m} \subseteq M.B\vec{m}}{\langle (B,A),M\rangle \Rightarrow_\Gamma \langle (B,A),M[A\vec{m}/\vec{m}]\rangle}\ \mathbf{App_1} \qquad \frac{\neg\exists \vec{m} \subseteq M.B\vec{m}}{\langle (B,A),M\rangle \Rightarrow_\Gamma M}\ \mathbf{App_2}$$

$$\frac{\langle P_1, M\rangle \Rightarrow_\Gamma \langle P_1', M'\rangle}{\langle P_1 + P_2, M\rangle \Rightarrow_\Gamma \langle P_1' + P_2, M'\rangle \ \&\ \langle P_2 + P_1, M\rangle \Rightarrow_\Gamma \langle P_2 + P_1', M'\rangle}\ \mathbf{Int}$$

$$\frac{\langle P_1, M\rangle \Rightarrow_\Gamma M \quad \langle P_2, M\rangle \Rightarrow_\Gamma M}{\langle P_1 + P_2, M\rangle \Rightarrow_\Gamma M}\ \mathbf{Int_T}$$

$$\frac{\langle P_1, M\rangle \Rightarrow_\Gamma \langle P_1', M'\rangle}{\langle P_2 \circ P_1, M\rangle \Rightarrow_\Gamma \langle P_2 \circ P_1', M'\rangle}\ \mathbf{Seq_1} \qquad \frac{\langle P_1, M\rangle \Rightarrow_\Gamma M}{\langle P_2 \circ P_1, M\rangle \Rightarrow_\Gamma \langle P_2, M\rangle}\ \mathbf{Seq_2}$$

Figure 1.3: The SOS of Gamma, described in terms of a 1-step transition relation on configurations. The transition relation $\Rightarrow_\Gamma$ is the least satisfying the above clauses. Rule **Int** has multiple conclusions, indicating that either conclusion can be deduced if the antecedent is true.

composition of two functions. `f1 ∘ f2` applies `f2` until it terminates (Rule $\mathbf{Seq_1}$). After `f2` terminates, `f2` is discarded and `f1` is applied (Rule $\mathbf{Seq_2}$). Sequential composition therefore behaves in a similar way to functional composition in an eager functional language [49] (hence the choice of notation). The second connective is written '+' and indicates the interleaved composition of two functions[4] For example, $f1 + f2$ will apply whichever of $f1$ or $f2$ are applicable to the multiset, if either of them can be applied (Rule **Int**). If both of them can be applied, one is chosen non-deterministically. If neither can be applied, the composition terminates (Rule $\mathbf{Int_T}$).

1.1.2 Examples of Γ programs

So, what can we do with all this? In this section, we attempt to whet the readers' appetite for MST programs with a number of 'classic' Γ programs. All of these have appeared in the literature (see, for example, [16] for a selection of programs).

An addition program

Imagine that we wish to write an MST program which adds up a set of numbers. In a conventional language (e.g. C [82]), we may start by considering in which sort of data structure we wish to represent our data. A linked list? An array? Then we would consider what sort of mechanism we wish to use to traverse the data structure that we have created. But, given a programmer's natural lazy instincts [141], together with our knowledge of the associativity of 'plus', what we would really *like* to say to the computer,

[4] In the literature, '+' is often described as a *parallel* operator, a terminology with which we disagree for reasons which will become apparent in Chapter 3.

if we could, would be 'just add up the numbers, in any way you like.' With this thought in mind, consider the following Γ program:

$$
\begin{array}{lll}
(B, A)\ \{\ldots\} & & \\
\text{where} & B(x, y) & = \text{True} \\
& A(x, y) & = \{x + y\}
\end{array}
$$

Reading this program from top to bottom, we realise that we have but a single primitive function (B, A), which is applied to the multiset. The primitive function is defined such that it takes two formal parameters x and y and then returns a singleton set containing their sum. In the process it consumes both arguments (functions always consume their actual parameters in Γ). With an initial multiset $\{1, 2, 4\}$, this program can reduce in three ways, which are shown in Figure 1.4. Each possible reduction path leads to the same final multiset. Each of these three reduction paths can readily be generated by using the SOS rules of Figure 1.3. The formal derivation of one possible reduction path is shown below:

$$
\begin{array}{lll}
\langle (B, A), \{1, 2, 4\} \rangle & \Rightarrow_\Gamma & \langle (B, A), \{3, 4\} \rangle \\
\langle (B, A), \{3, 4\} \rangle & \Rightarrow_\Gamma & \langle (B, A), \{7\} \rangle \\
\langle (B, A), \{7\} \rangle & \Rightarrow_\Gamma & \{7\}
\end{array}
$$

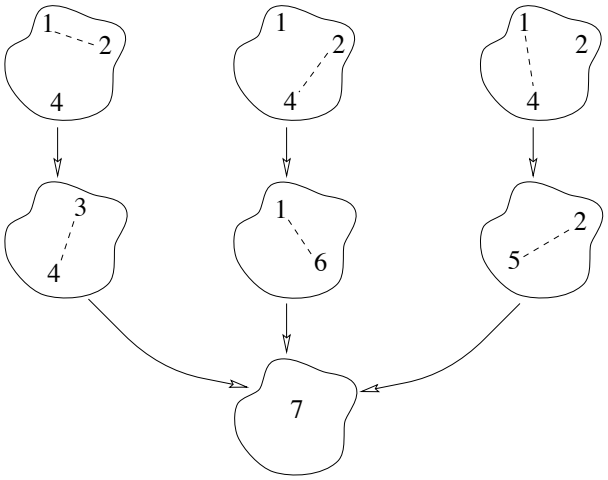


Figure 1.4: The three possible reduction paths of the addup program, applied to a multiset $\{1, 2, 4\}$.

A sorting program

The following Γ program sorts a list of elements, represented in the multiset as a set of (index, value) pairs.

$$
\begin{array}{lll}
(B, A)\ \{\ldots\} & & \\
\text{where} & B((i1, v1), (i2, v2)) & = \quad i1 > i2 \wedge v1 < v2 \\
& A((i1, v1), (i2, v2)) & = \quad \{(i1, v2), (i2, v1)\}
\end{array}
$$

The program searches the multiset for pairs of elements whose values are out of order with respect to the indices. The values of all such pairs are swapped. Once all of these pairs of elements have been swapped, the multiset represents a sorted list. An example reduction path for this program, can be seen in Figure 1.5.

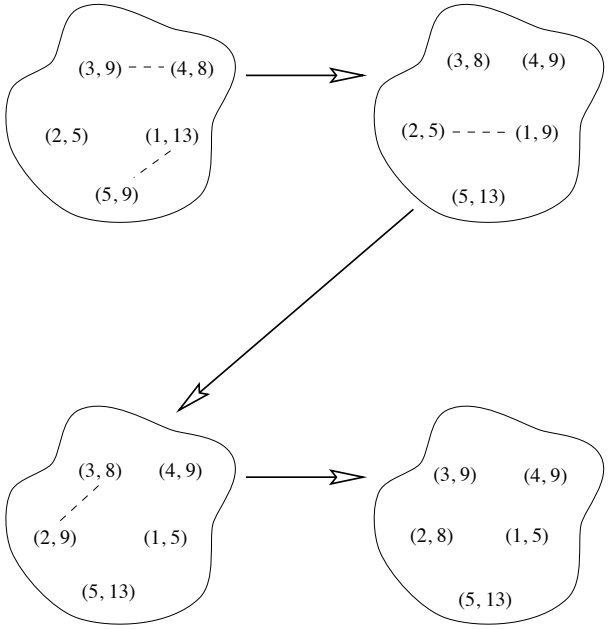


Figure 1.5: A possible reduction path for the sort program, applied to a multiset $\{(4,8),(2,5),(3,9),(1,13),(5,9)\}$. The example follows [16].

The sieve of Eratosthenes

A primes sieve on a multiset $\{2 \ldots n\}$ can be written in Γ as follows:

$$
\begin{array}{lll}
(B, A)\ \{\ldots\} & & \\
\text{where} & B(x, y) & = \quad x\%y = 0 \\
& A(x, y) & = \quad \{y\}
\end{array}
$$

Where $x\%y$ is the remainder of an integer division of x by y. The program proceeds by eliminating all of the numbers in the multiset which are multiples of other numbers in the multiset. Eventually, the only numbers left in the

multiset are those which cannot be divided by any others i.e. are prime. An example reduction path for this program is shown in Figure 1.6.

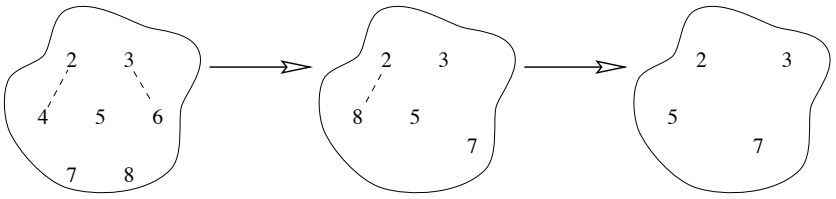


Figure 1.6: A possible reduction path for the sieve program, applied to a multiset $\{2 \ldots 8\}$. The example follows [16].

Much of the work done on Γ has attempted to generate semantics for the language and thereby to prove properties of the language. Of particular interest are [67, 48, 66]. Implementation work on Γ is described in [70, 139, 62, 39]. Some non-trivial applications of Γ (a UNIX kernel and image processing), are described in [34, 40]. An interesting Higher-Order variant of Γ is to be found in [100]. Many more relevant papers can also be found in [1].

1.2 The friends of MST

One of the strengths of MST is that it appears, more or less independently, in many different areas of computer science. While this does not *per se* mean that it is a significant development, it certainly causes one to believe that it addresses issues which are widely relevant. The fields in which MST has its strongest impact are illustrated in Figure. 1.7. Each of the relationships shown in the diagram, is discussed in more detail in the following subsections. Coordination languages have already been discussed, so the material is not repeated here. By calling the readers' attention to these apparently disparate evolutions of MST, we hope to convince her[5]. that MST is, indeed, a subject worthy of study in its own right. Additionally, we explore in greater detail in later chapters relationships between some of these areas, so a compass might aid the reader in following the trail we have laid out here.

1.2.1 Parallel and distributed computing

Below, we discuss the relationship between MSTLs and parallel programming. Most MSTLs (e.g. Γ [39, 62, 61]) are claimed to be amenable to parallel implementation; the intuition being that an unstructured state can easily be partitioned over a number of nodes. We shall see in due course

[5]Throughout this work, we refer to third parties as 'her', for formally motivated reasons: $\{s, h, e\} \cup \{h, e\} = \{s, h, e\}$. 'Her' follows from 'she'. (Notice that this argument works in English, which is why this thesis is written in English.)

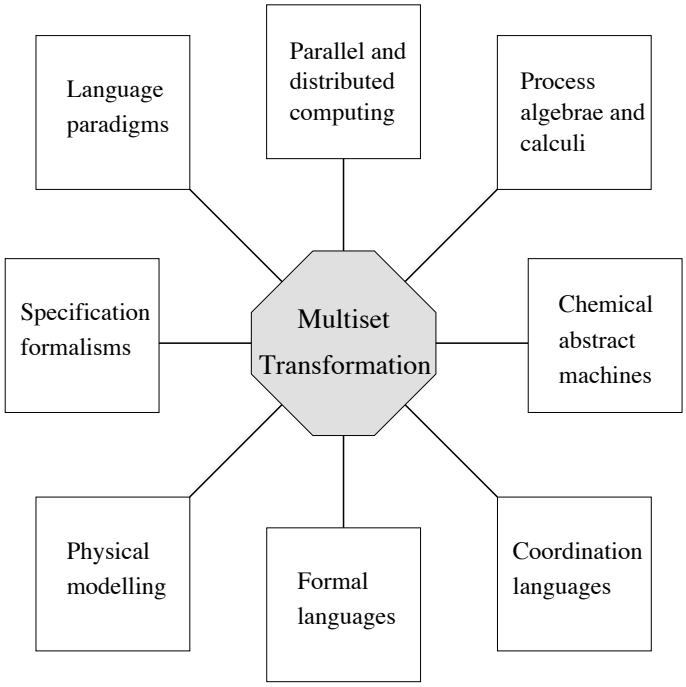


Figure 1.7: MST's closest colleagues. A strong relationship between MST and another area is indicated by a line. The connections between each of these areas and MST are explained in this chapter.

that, at least for certain languages, this claim of almost free parallelism is a mistake. The difficulty arises because of the sequential dependencies on function applications required by the multiset elements and the functions' conditions. We therefore appear to lay to rest the dependencies beneath a shroud of parallel operators. However, once the program is executed, these dependencies leave their grave and rise again.

Linda

Linda [59, 58] is undoubtedly the most famous MSTL. Proposed by Gerlernter and Carriero, Linda is a set of primitives designed to be added to conventional programming languages to enable coordination through interoperability of heterogeneous software components. No control-flow coordination is specified (all processes run in parallel), while intercommunication between the components is enabled through provision of a shared tuple space (a multiset). Tuples can be inserted into the space using the `in()` primitive, removed from the set using the `out()` primitive or read (that is, a tuple is observed but not removed from the tuple space) using the `read()` primitive. Removal or reading of a tuple is performed via pattern-matching on tuple

elements. Linda also includes an `eval()` function which performs some action which eventually places a tuple in the tuple space. For example, `eval()` may spawn new processes which eventually deposit their results in the tuple space. Tuples placed into the tuple space remain until removed.

Communication in Linda has the curious property that neither sender nor receiver have to know where the other program is in time or in space. In particular, it is possible to leave a message in the tuple space for a program which has not started to execute. When another program starts to execute, it can retrieve, from the tuple space, the message which was left for it earlier. This very simple but very basic model of communication was christened *generative communication* by its creators.

Linda has been implemented on a number of architectures and is used in a number of projects. It is even available commercially. More details of Linda, its programming style and applications can be found in [33, 31, 32].

Bonita

Rowstron and Wood's Bonita [121] is an alternative set of primitives to Linda's, into which it is possible to translate Linda's primitives. In Linda operations, the `in()` and `read()` operations are synchronous: they wait until an eligible tuple is returned. Experience with implementation and applications of Linda systems convinced Rowstron and Wood that inefficiencies in distributed Linda implementations stem not from the time required to access the tuple space but from the time taken to send messages over a network. Bonita's primitives are asynchronous, allowing further computation during the time that a process must wait for a requested tuple to be returned from the tuple space.

1.2.2 Specification formalisms

Most MSTLs possess a simple operational or denotational semantics. Possession of such a semantics enables formal reasoning about the language and about programs written in the language. The availability of formal reasoning makes it possible, for example, to prove the correctness of a particular program relative to that program's specification. A well-defined mathematical semantics for a formalism also makes it possible that a program be subject to successive *refinement* in a correctness-preserving way [5, 107, 106, 6, 11, 12, 7, 13]. It is encouraging that many MSTLs have actually been used as specification languages in their own right. Examples include UNITY [35], DisCo [87] and Action Systems [8, 9]. All of these formalisms are described below. These three specification languages perform computations based on the chemical-reaction metaphor: functions continue to be applied to the state until application of none of the functions can change the state any more.

UNITY

UNITY [35] is a specification formalism intended to be used to write correct parallel programs. Programs consist of two parts. One is an initialisation

statement for each variable which appears in the program. The other part of the program is a set of conditional assignment statements, which can be executed in any order (or in parallel). The only constraint is that imposed by a fairness constraint: every statement is executed infinitely often. At each computation step, some statement is chosen and executed. The program never terminates, although there may come a point when no assignment causes alterations in the state. The state has therefore reached a fixed point with respect to the program, and can therefore be treated as the result of the computation.

A point to note is that the stability condition for UNITY programs is different from that of Γ programs. In Γ, a multiset M is said to be *stable* with respect to a program P iff P cannot be applied to M. In UNITY, however, a multiset M is defined as *stable* with respect to a program P iff applying P to M does not change M. An example of this can be seen if we imagine the function

$$\text{if } x = 1 \text{ then } x := 1$$

When applied to a state $\{x := 1\}$. In UNITY, the program is immediately stable whereas, in Γ, a textually similar program will *never* be stable with respect to the state. In Chapter 2, we discuss in greater detail stability of programs with respect to a state.

Action Systems

Action Systems [5, 9, 10, 11, 131, 87, 30] is a specification framework for parallel reactive systems. An Action System consists of a number of processes, executing in parallel. As in UNITY, statements are guarded commands. Furthermore, also in common with UNITY, the only restriction on the order of process executions is that imposed by a fairness assumption. In Action Systems, abortion is favoured over all other possible reductions. That is, if one of the operands of an interleaving composition will abort if executed, then abortion must occur.

DisCo

Another language from the same stable as UNITY and Action Systems is DisCo (*Dis*tributed *Co*operation) [77, 76, 85, 87, 86]. The reactive nature of its application domain motivates use of Lamport's Temporal Logic of Actions (TLA) [88] as a logic for formulating and for proving program properties. DisCo has been used for a number of small- to medium-sized applications [76, 85] and for specification of object-oriented (OO) reactive systems [77]. In all these applications, the strength of the model is shown through its amenity to stepwise refinement of specifications into executables.

Transaction-Based Programming and Schedules

The language TBP (Transaction-Based Programming) was introduced in [45] as a specification language for parallel programs. Similar in orientation to

Γ, the work on TBP was carried out independently from that of Γ, until the work of Chaudron [36] (described below). To TBP was added the language of Schedules, which allows the programmer (or a tool) to specify orderings on TBP component executions. The TBP program describes *what* is to be done, while the schedule describes *how* it is to be done. An executable program therefore consists of two parts: the original TBP program and the schedule. Using stepwise refinement [107], a lower-level schedule can be derived from a higher-level schedule, until the schedule is tuned for optimal execution on the desired architecture.

Chaudron [36] has used Schedules as a coordination language for Γ. He has produced a number of impressive results showing how well Schedules can be applied a number of Γ programs, including programs for matrix transformation. Chaudron's work differs from much of that done on Γ in that he uses Γ as a *programming* language, rather than a coordination language.

Of particular relevance to the current work are:

1. Schedules features one-shot functions with explicit recursion. This is the approach taken in Chapter 2 of this thesis, which differs from that taken for Γ and for CGP. The latter two formalisms feature implicitly recursive functions and no one-shot functions.

2. Schedules' parallel operator is a true, non-prescriptive, parallel operator. Again, this differs from the approach taken in Γ and CGP, in which the 'parallel' operators are interleaving operators. We give example prescriptive and non-prescriptive parallel operators for multiset transformation languages in Chapter 3.

By separating the high-level program description from the low-level descriptions, Schedules encourages the derivation of multiple versions of a program, each tuned for a specific architecture. Throughout, the original program is preserved, making it easy to see which refinement steps have been made for the architecture in question.

1.2.3 Process algebrae and calculi

Two areas of vital importance to the development of the semantics of program operators are those of process algebra and process calculi [101, 69, 15]. Process algebra, as its name suggests, adopts an *algebraic* approach to program (process) composition operators, leaving operational considerations to be derived from the axiomatic description. Each process composition operator is defined by a set Σ of function symbols together with their arities and a set E of equations over terms over σ. Process algebra has always been of fundamental importance to formal approaches to parallelism, although neither field is impotent without the other. In process calculus, we start with an operational semantics, which is then used to derive program equivalences. Despite these differences, the two approaches are sufficiently closely related to warrant a joint presentation. Historically, the similarities have also been great, with CCS (a process *calculus*) motivating the development of ACP (a process *algebra*).

Arguably the first approach to concurrency theory which interested itself in algebraic issues was Milner's Calculus of Communicating Systems (CCS) [101, 102]. Offering an operational semantics to his operators, Milner showed how to prove the existence of certain algebraic laws. Once this was achieved, the discussion could shift from a purely operational perspective to a focus on the algebraic issues thereby raised. Since then, Milner has developed the π-calculus (first introduced in [103, 104]), a more general form of CCS. At about the same time as CCS was introduced, Hoare introduced his Communicating Sequential Processes (CSP) [69], which was based upon a trace semantics. Later, the name TCSP ('Theoretical CSP') was introduced to describe a version based upon failure semantics [29]. Confusingly, it is now common to refer to TCSP as CSP. Process algebra was introduced by Bergstra and Klop, in the form of the Algebra of Communicating Processes (ACP) [26, 23]. Instead of beginning with an operational approach from which algebraic laws were derived, ACP began with a collection of laws which 'should' be satisfied by any operational semantics. Of the four formalisms here mentioned, Theoretical CSP is the most abstract, while CCS, the π-calculus and ACP are more operationally oriented [14]. That is, CSP identifies more processes than the other two and is more of a specification language than the other two. We briefly introduce each of these formalisms below. More details about process algebrae can be found in [101, 69, 15]. The sections on CCS and CSP follow [15], while the section on ACP follows [23]. We give merely the briefest appetiser of each formalism, to whet the readers appetite.

Each of the process algebrae/calculi described here offers a collection of process composition operators, together with either a set of equations describing the processes identified under the algebra, or an operational semantics from which such equations can be derived. Many variants exist of each of these formalisms, each with its own added constructs.

ACP

ACP possesses a set A of so-called *atomic actions*, including a constant δ for deadlock. The process composition operators include nondeterministic choice, sequential composition and interleaving composition. Nondeterministic choice $+$ chooses between its operands (note: $+$ has a completely different meaning in Γ, where it indicates the interleaved execution of its operands [65]). Sequential composition of processes x an y is written $x.y$, or just xy. Interleaving composition $|||$ interleaves execution of its operands. It is defined in terms of $\lfloor\!\lfloor$, which is the same as $|||$ except that $x \lfloor\!\lfloor y$ takes its first action from x. $|||$ in ACP therefore corresponds (intuitively) to $+$ in Γ. ACP possesses a communication function $\mid$ such that if $a \mid b = c$, then c is the action that results from simultaneously executing a and b. Processes share actions rather than data [23, page 2]. $x \mid y$ is like $x ||| y$ except that the first action has to be a communication between the first step of x and the first step of y.

Once an algebraic description of a set of operators is given, an operational semantics for the operators can be derived [14]. This is not always a

straightforward task, as the discussion of 'weight' in [24] demonstrates.

CCS

For every atomic action in CCS, we have precisely one other atomic action with which it communicates. Thus, we have both a set Δ of names and a set of *conames* $\overline{\Delta} \stackrel{\text{def}}{=} \{\overline{a} : a \in \Delta\}$. When a and $\overline{a}$ communicate, the result is a *silent step* τ. Therefore, the set A of actions is defined as $A \stackrel{\text{def}}{=} \Delta \cup \overline{\Delta}$ and the set A_τ of actions including τ is simply defined as $A_\tau \stackrel{\text{def}}{=} A \cup \{\tau\}$.

The nondeterministic choice of CCS is the same as the $+$ of ACP. CCS has no general notion of sequential composition. Instead, only *prefix* composition is possible.: if $a \in A_\tau$ and x is a process, then ax is a process. CCS's $|$ operator is similar to the $|||$ operator of ACP, but with more restricted communication possibilities. In CCS, if $x = t(x)$ is an equation, then $\mu X.t(x)$ is a process satisfying this equation. In our semantics work in Chapters 2 and 3, we make use of the μ-notation to describe recursive programs.

The π-calculus

The π-calculus [103] is a CCS-like formalism in which communication of *names* occurs through named *channels*. A communication over a channel can only occur if the receiver names the same channel as an input as is named by the sender as an output channel. Four constructions are added to this basic notion.

Firstly, sequential composition is prefix composition, as we have already seen for CCS. Secondly, $P \mid Q$ is an interleaving composition of P and Q, in which communication *may* take place. Thirdly, $!P$ is a replicant of P: P can be copied as many times as necessary. Finally, $(\nu x)P$ restricts the scope of name x in P to P. That is, occurrences of name x outside P are regarded as different from those inside P.

CSP

In CSP, a process a can only communicate with itself, an action which leaves a unchanged.

CCS possesses two kinds of choice operator: external choice $[]$ and internal choice $\sqcap$. The former can be influenced by the environment (hence its name). The latter cannot be influenced by the environment and occurs whenever the choice is determined by hidden actions. $x \sqcap y$ can be represented in ACP and CCS by $\tau x + \tau y$, while $[]$ cannot be directly represented.

Furthermore, like CCS, CSP has prefix composition. CSP has two operators for parallel (interleaving) composition. $|||$ is interleaving without communication (like $\|$ in ACP) and $\|$ is communication without interleaving. Like CCS, CSP offers recursion in the form of μ-operators. The interpretation, however, is different from that in CCS.

The link between MSTLs and process algebrae/calculi

With their goals the exploration of alternative language constructs and their interactions, process algebrae and calculi are close in spirit MSTLs such as Γ. Consider, for example, the following, which was stated in [65, page 344].

> For the sake of modularity, it is desirable that a language offers a rich set of operators for combining programs. It is also fundamental that these operators enjoy a useful collection of algebraic laws in order to make it possible to reason about composed programs.

This similarity of purpose manifests itself in the willingness to add new operators to an existing language to study their effects on that language. (compare [14] and [65]). We shall see echos of ideas from process calculi (particularly with regards to deadlock and non-deterministic choice) in later chapters.

Although possessing of a rich shared heritage, MSTL and process algebra also orient themselves somewhat differently. The major difference in the orientation taken in the MSTL world from that taken in the process algebra world is that, in the latter, the *algebraic* laws define an operator, from which laws an operational semantics may be derived. For the former (MST), the semantics is taken as defining the language and the algebraic laws are derived. Although this difference might appear slight, it has some ramifications. For example, it has been shown that it is not possible to define a finite axiomatisation for the free-merge (interleaving) operator, without making use of an auxiliary operator [105]. For this reason, [22] introduced a so-called left-merge operator, in terms of which the free merge can be given a finite axiomatisation. The fragment of the process algebra term-rewriting system corresponding to interleaving therefore contains two sets of rules: one for the free merge (which are defined in terms of the left-merge) and one for the left-merge itself. Thus is the algebraic difficulty of a finite axiomatisation manifested operationally by the presence of several operators. In comparison, operational semantics of interleaving operators in MSTs require no auxiliary operators [66]. Another example of the effects of regarding the algebraic laws as fundamental can be seen when we consider that adding an operator to a process algebra potentially involves adding equations to describe the interaction between programs containing the new operator and each of the old operators. In general, standard operational semantics of similar operators (sequential composition, interleaving, deadlock etc) require no such additions to be made when they are added to a language.

Another similarity between process algebrae/calculi and MSTLs is highlighted in the following section, where we describe a single model, an extension of Γ, in which semantics for process calculi can be given (among other things).

1.2.4 Chemical abstract machines

The Chemical Abstract Machine (ChAM) [25, 27] was introduced as a generalisation of Γ. In the ChAM, molecules are treated as terms of algebrae and may be *heated* to break complex molecules apart or *cooled* to allow complex molecules to form from simpler molecules. Furthermore, the ChAM introduces the notion of a *membrane* which isolates some part of the chemical solution. Within the membrane, reactions continue independently of the rest of the solution. A limited form of membrane permeability is introduced: molecules are able to pass through the membrane by passing through an *airlock*.

ChAMs can be seen as a unifying concept shared by MSTLs such as Γ and by process calculi and algebrae [15]. On the one hand, the ChAM is avowedly an extension of the Γ model and clearly shares many of the same properties. On the other hand, the ChAM has been used to give semantics for a number of process calculi, including CCS and the π-calculus [27].

The ChAM makes a much more literal interpretation of the chemical reaction metaphor than that taken in Γ. Interestingly, we shall see another very literal interpretation of the metaphor within a rather different context in Section 1.2.7. There, we shall see chemical reactions seen not as a means of giving a semantics to concurrency, but as a way of investigating the origins of life.

1.2.5 Language paradigms

Conventional (imperative) languages are based upon the Von Neumann computation model, in which a list of instructions is given to the computer, which executes each in turn. Over the last few decades, interest has grown in computation based upon models other than Von Neumann's. The most important of these new models are declarative languages, by which we mean both functional [49] and logic languages [71]. Declarative languages offer a model in which the machine is not explicitly given a list of tasks to perform in a particular order. Instead, the emphasis is on removal of the operational aspects of program execution (slogan: "How the solution is to be reached.") and, instead, towards providing a (mathematical) description of the problem to be solved, the logical structure of which allows the generation of solutions (slogan: "Describing the problem.").

In a logic language [71, 75], a program consists of a set of terms from a restricted first order propositional logic theory with equality (i.e. Horn clauses [71]). Computation proceeds by generating inferences from the set of terms until either the goal or its negation are generated. It may be possible for reduction to proceed non-deterministically or in parallel [128]. We will not further discuss logic languages in this thesis.

Functional languages treat computation as reduction of terms of the lambda calculus [21]. Examples of functional languages include Haskell [135], Sisal [54], the ML family [112, 119] and Miranda[6] [134]. A functional language consists of syntactically-sugared lambda terms, which provides a far

[6]Miranda is a trademark of Research Software Ltd.

richer set of primitives than that provided by pure lambda calculus. Programs are written such that the entire program is actually a single lambda expression [49]. Computation consists of reductions of the lambda term until a normal form is reached (actually, most functional language implementations terminate execution of a program once a weak head normal form (WHNF) is reached [49, 64]). As in the case of logic languages, it may be possible that several parts of the term can be reduced in any order or even in parallel [113, 42, 43].

MSTLs live in the space between imperative languages and declarative languages. Like declarative languages, they describe the 'what' rather than the 'how' of a program by omitting many of the implementation details of a program. For example, the Γ `sort()` program of 1.1.2 can be used to generate a straightforward logical definition of a sorted list, by simply negating the reaction condition on the single primitive function. That is, we can define a sorted list L as:

$$\neg(\exists(i1, v1), (i2, v2) \in L.i1 > i2 \wedge v2 \leq v2)$$

A difficulty with current declarative languages

Actually, the situation is even curiouser: for many problems, an MST program is *more* declarative in style than a naïve functional program, because the unstructured state imposes no requirements on the order of function application to data elements. To illustrate this point, we compare different versions of programs which add a number of integers. We give a typical functional-programmer's solution, written in Miranda. The numbers are placed into a list and passed to function sum.

```
sum [...]
    where sum nil = 0
          sum (head:tail) = head + (sum tail)
```

This program should be read as follows: recursively traverse the list, adding each number to the total as you come to it. Stop when the list is empty.

Compare this to the Γ program of Section 1.1.2. This program will repeatedly add pairs of numbers, placing their sum into the multiset. Only when the multiset is of cardinality one will the function be inapplicable, leading to termination of the program. The remaining element in the multiset is the sum of the original elements. Notice that, unlike the functional program, the multiset version of the program imposes no ordering on removing elements from the multiset. Indeed, the program can even be executed logically in parallel.

In the light of our example, we can clarify our previous observation that Γ's programming style is more declarative than that of a declarative language. We must write a functional program in terms of data structures which impose extra-logical sequentiality on the ordering of function applications. This requirement introduces algorithmic aspects into what is, allegedly, a declarative program.

1.2.6 Formal languages

Another application which suggests itself is using MSTs to generalise formal languages [124, 125]. Formal languages describe the sets of strings generated by recursively applying rules to initial strings. With MST and, in particular, Γ, we have a notion of applying functions recursively to an initial multiset. All the possible multisets which could be generated by this process would be regarded as 'words' in the 'language' defined by the rewrite rules. The connection between formal languages and Γ is accentuated by the similarities between a set of production rules in a formal language and an interleaving composition of (recursive) primitive functions in Γ. So far, the only serious work which has been done in terms of understanding the connection between MSTLs and formal languages is in the area of Lindenmayer systems (L-systems) [91].

Lindenmayer systems were proposed as a formal model of plant development. An L-system consists of a set of conditional rewrite rules (productions) which are applied to a string of symbols. All elements of the string are rewritten at every production step: if no production applies to an element of that string, then the identity production is applied. L-systems have the curious property that a topological tree (the 'plant' being grown) is embedded in the string. For this purpose, matching pairs of angle brackets are added to the string to indicate the beginnings and ends of branches.

For more sophisticated models of plant growth, context-sensitive L-systems were developed [115]. These are known as IL-systems, where the 'I' stands for 'interactive'. In IL-systems, rewrites are carried out in a 'data context', which means that some substring of the string is examined in addition to the element to be rewritten (the *predecessor*). The data context has to be local to the predecessor in the sense that the data context has to straddle the predecessor or be adjacent to it. Using IL-systems, much more complex models of plant growth can be built, incorporating propagation of auxins (plant hormones) through the growing structure to alter its development [115].

For sophisticated plant models, we have seen that data context and logically synchronous rewrites are needed. These we discuss, in terms of multiset transformation, in Chapter 3. Good overviews of L-systems can be found in [122, 123, 116, 118].

1.2.7 Physical modelling

MST languages frequently exhibit nondeterministic behaviour as a result of the multiset's lack of structure and the parallel (or interleaved) computation model which accompanies it. In fact, as we have already remarked in Section 1.2.6, many MST programs look more like production systems such as OPS5 [38] or grammars [124] than conventional computer programs. This apparent family resemblance [144] led some to wonder about MST's applicability for modelling physical phenomena. Typically, we could imagine modelling individual objects such as particles as elements of the multiset: interaction could be described by a number of production rule-like functions, executing in an interleaved or parallel fashion. We examine some of the pos-

sibilities below, after giving some more motivation for the orientation here suggested.

Accurately modelling physical systems using computers requires that appropriate abstractions of physical phenomena be made: computers' discrete nature, coupled with their finite resources and limited execution speeds, makes direct solution of many numerical models intractable [57]. Therefore, explicit simulation of system's evolution is often undertaken [120]. However, whenever abstractions of realistic physical situations are made in the name of making a model computationally feasible, the question arises as to whether and how the abstractions made affect the simulation results obtained. To ameliorate this potential problem as much as possible, it is therefore desirable that the simulation's model be as close to the physical reality as possible. We can easily grasp the intuitive appeal of a model in which physical laws are implemented as functions, physical objects (or parts thereof) are modelled as elements of the multiset, and very little explicit control-flow has to be encoded in the program: every function is merely applied to all particles logically in parallel, or in an interleaved fashion. After all, in nature, it would appear that 'sequentiality' indicates the presence of dependencies between different processes rather than a 'formal' requirement that one process terminates before another starts.

AlChemy

A fascinating example of this kind of model is presented in the work of Walter Fontana and Leo Buss [51, 50, 52], who build abstract models of chemical reactions in order to understand how complex molecules could arise out of collections of simpler models. Their model makes use of an abstract, unstructured, multiset of λ-terms [21] which play the rôle of the molecules. At every time step, two lambda terms are selected at random from the solution (the multiset). These two 'molecules' are then made to 'react' with each other. The reactions are modelled by function application and a slightly modified form of β-reduction [21]. The reaction products are placed back into the multiset. The cardinality of the multiset is kept constant by an analogue of a flow reactor [129], which arbitrarily throws elements of the resulting multiset away until the desired cardinality is reached. Comparisons are made between the results obtained when the reactions are 'catalytic' in the sense that the parents of the reacting molecules are placed back into the multiset and when the reactions are not 'catalytic'. Furthermore, investigations are undertaken into multisets of terms which are auto-catalytic. Auto-catalysm is the ability of solutions of molecules to regenerate themselves when some of the molecules are removed. This is achieved through the complex and mutual catalytic relationships between the different substances, ensuring that missing substances can be regenerated through reactions between the remaining substances in the solution. Auto catalytic solutions are therefore self-maintaining, a crucial property for living organisms and the reason why they are so interesting for origin-of-life researchers.

We show how to implement AlChemy in an MSTL in Chapter 2.

Cellular automata

Another model which is similar to L-systems is Cellular Automata [145]. A cellular automaton can be defined in many ways (see, for example, [136] for a good overview), so here we will merely sketch a definition, due to Wolfram [145]. A CA consists of an n dimensional array of *cells* and a set of rules governing how the state of a cell at time $t+1$ is related to the states of its neighbours and itself at time t. We leave unspecified which cells are considered neighbours: clearly, there are several possibilities. According to Wolfram, rules have to be symmetrical in the sense that any orientation of a predecessor (sub-)state produces the same successor (sub-)state. In other words, rotations of the space in which our rewrites occur does not effect the results of the rewrites. In terms of physical concepts, this requirement is required to guarantee isotropy and homogeneity in the automaton's evolution [145, Page 8, footnote]. Additionally, symmetry between the treatment of zeros and other integers is broken by requiring that any configuration of zeros cannot be rewritten to anything apart from zeros. This condition is known as a 'quiescence' condition: non-zero elements cannot spontaneously arise. In terms of physical intuitions, the quiescence condition demands that nothing can come into existence out of nothing. Interestingly enough, Sands' remark that Γ programs cannot rewrite non-empty multisets and terminate [126, 127] seems spiritually related to this aspect of CAs. In a similar vein, L-systems do not allow productions to produce successors from empty predecessors, for precisely this 'philosophical' reason.

At every time step in a cellular automaton's evolution, all cells are updated by applying the relevant rule to them (usually, only one such rule is applicable to each cell). The result is a logically synchronous parallel update of all cells at every time step. An example evolution of a five state automaton can be seen in Figure 5.15 in Chapter 5.

To relate CA to L-systems, we can see that L-systems are a generalised case of 1-D cellular automata. There are at least four differences between the two models:

1. CA rewrite rules possess rotational symmetry: the rules have to have (in the language of L-systems) identical left and right data contexts.

2. Quiescent conditions are not required for L-systems: rules can rewrite anything to anything.

3. In CAs, the size of the context (predecessor set) required for each rewrite is determined globally. In L-systems, it is determined locally. This flexibility on the part of L-systems makes it possible for the predecessor sets for different productions in L to be different sizes.

4. L-systems often include symbols which are 'invisible' during context determination, but are used for interpreting the string to produce a graphical output. Such 'phantom' elements (they cannot be seen and yet they exist) do not exist in CAs.

5. The boundary conditions for CAs are different from those of L-systems. In L-systems, the string grows and shrinks as productions are applied to it. No circular boundary conditions are possible: the string has a definite beginning and a definite end. In CA, the situation is somewhat different. Rewrites take place in either (i) a logically infinite array or (ii) a finite array with some boundary conditions (often circular). In neither case does the array itself change size as rewrites progress.

We mentioned earlier that a plethora of definitions exist for cellular automata. The variants include Lattice Gas [55], asynchronous CA [92], Dimer automata [130], Ising systems [93] and Diffusion-Limited aggregation [143]. Each of these alternative systems has its own particular characteristics, be they in the form of the array or in the mechanisms for rule application. Below, we discuss diffusion-limited aggregation models in more depth, because we use DLA as an example of probabilistic multiset transformation in Chapter 4.

The similarity of rules in CA to primitive functions in an MSTL lead one to believe that the former can be encoded in the latter. This is, indeed, the case, as we shall see in Chapter 5.

Diffusion-limited aggregation

Diffusion-limited aggregation (DLA) was introduced in [143] and has been used to describe a number of accretive growth phenomena, including ice crystal formation, mold growth and dielectric breakdown [111]. DLA is actually a form of probabilistic cellular automata (CA), which are described above. The diffusion-limited aggregate begins with a single occupied site on a lattice. Beginning at random positions on a circle with radius b centered on the occupied site, a particle begins a random walk. Such particles are known in the literature as 'walkers'. If a walker crosses a circle of radius $d > b$, centered on the occupied site, then it 'dies' i.e. is removed from the lattice. If it is adjacent to an occupied site then it accretes onto the growth form with a probability which can be intuitively interpreted as the 'stickiness' of the accreting material. Once the current walker has either died or accreted onto the growth object, a new walker is released. The process continues. As more walkers accrete onto the growing form, the more the form takes on a filamentous shape called a DLA cluster with a box dimension [47] of 1.67. More work on DLA clusters is described in [140, 80].

Genetic algorithms

Genetic algorithms (GA) were introduced as an optimisation technique based upon a metaphor of natural selection within a gene pool. A genetic algorithm (of which there are many variants, see [44] for a survey) takes a multiset of 'genetic material', normally represented as strings of equal length. At each iteration, sexual reproduction is mimicked by choosing at random two of the strings (the genetic material of the mates). A point within one string is chosen at random and the material from that point in the string is swapped

with the material from the corresponding point in the other string. This stage is known as 'crossover', again using the genetic analogy. The resulting string is placed back in the population with its 'parents'. Finally, one string is chosen from the multiset and discarded, representing the death of one of the organisms in the population.

In some variants of GAs, occasional 'mutations' alter the values of single or multiple parts of a string. Such 'mutations' enable the GA to explore a larger part of the search space than would otherwise be possible [63].

For selection purposes, each string is assigned a 'fitness', such that fitter strings are more likely to be chosen than less fit strings. A normal implementation of a genetic algorithm proceeds by calculating the sum of the fitnesses of all the strings and then, for each string i, allocating a probability p such that:

$$p_i = \frac{f_i}{\sum_{k=1}^{n} f_k}$$

where f_i is the fitness associated with string i.

GAs are related to MSTLs in that they transform a multiset of genetic material. The probabilistic selection of members of the population for each sexual reproduction step has (in part) inspired the work on probabilistic MST, described in Chapter 4. Genetic Algorithms and their implementation in an MSTL are discussed in Section 1.2.7 of Chapter 4.

1.3 Open problems in MST

In the previous sections, we described a number of the areas in which MST models have been developed or, at least, seen to be useful. Our intention was to show how compelling MST is in a variety of different areas. In every case, striking similarities exist between the different MST models and languages. So, it would appear natural to assume that MST models fulfill some important function for many fields. With this survey giving us motivation, we claim that we can satisfy the reader who wishes to ask 'why do you use MST at all?'. Furthermore, all of these areas have influenced the work reported in this thesis, which hopefully helps to ensure that this thesis reports results of relevance to all of the areas mentioned above. Finally, the unified model for parallel transformation presented in Chapter 2 of this work is able to capture a large number of the formalisms here described: a testimony to its power and applicability.

Unfortunately, MST is not yet a mature discipline: much work remains to be done. Some of these open problems relate to particular languages (mainly Γ) while others are more general problems which one would expect to arise whenever multisets are used as data structures. We claim that several of the difficulties are genuine bugs in particular languages; other problems are only evidence of areas which have not yet been fully investigated.

This thesis tackles the problems described below, except for that of efficiency, although it does not claim to provide a final answer to each of those it does address.

1.3.1 Efficiency

Arguably the most pressing problem with multiset transformation languages is that of efficiency[7]. When a function is applied to the multiset, a search has to be performed for elements which satisfy the desired properties. Efficiency issues have been addressed for Linda [58] implementations, with reasonable success [142]. However, most MSTs allow that more than one element be removed from the multiset at the same time, causing the basic complexity of access to a multiset to rise. One might be tempted to think that the inefficiencies can be reduced by altering the control-flow behaviour of the program so that functions which cannot be applied are not even tried. For example, in order to gain efficiency (perhaps for particular architectures), mechanisms are proposed whereby interleaving operators can be replaced by sequential operators, and vice-versa [65, 66, 67]. Actually, these solutions do not address the problem: the inefficiencies arise because of the multiset search itself. Altering the control-flow behaviour of a program only slightly mitigates the problem, by reducing the constants associated with the complexity. The basic complexity stays the same. Consider the following argument:

The complexity of applying a single primitive function to the multiset is O(n!/(n-a)!), where n is the cardinality of the set and a is the arity of the function. For a binary function, we therefore have a complexity of $O(n^2)$ for a multiset of cardinality n. In a sum of primitive functions, the complexity of termination detection is *O(n!/(n-a)!)*, where a is the *largest* arity of any of the primitive functions.

Replacing the parallel composition with sequential composition saves only a factor 2 (if the functions have the same arity) which is insignificant compared to the complexity of applying either of the primitive functions to the multiset. Therefore, the problem of efficiency has to be tackled by reducing the cost of the multiset search. One possibility is to give the multiset structures. We say very little about multiset structuring in this thesis, although some work can be found in [97, 53].

1.3.2 Compositionality

For a programming language to be usable, its constructs have to behave identically when placed in larger contexts: the properties of the whole program must be deducible from the properties of its parts. This property is called *compositionality*. Unfortunately, a number of multiset transformation languages lack this property, leading to the programmer's job being made much more difficult than it might otherwise be. A number of examples of non-compositional behaviours in MSTs are well known [37]: these and other curiosities can be found in Chapter 2.

[7] Where *efficiency* is defined as the ability of a program to terminate before the person who wrote it.

1.3.3 Flexibility

Another issue of enormous importance is that of flexibility. Coordination languages must make minimal assumptions about the programs they coordinate in order to be as widely applicable as possible. This is manifestly not the case for some MSTLs. We give some examples below.

Non-atomic primitive functions

The Γ model assumes that all primitive functions are atomic. When combined with the semantics of interleaving composition (which states that a composition of programs applies at most one of the components to the multiset at every execution step), we notice that all functions must block while they are applied to the multiset. If functions are significant pieces of code, their blocking will easily negate any advantages won by executing programs in parallel. To avoid this potential pitfall, we need mechanisms for establishing asynchronous state update and access for larger programs, one candidate for which is discussed in Chapter 3.

Data contexts

Many programs require the ability to examine parts of the state without trying to update it. Examples are given in Chapter 3. Most MSTLs, such as Γ [65] and CGP [37] do not provide such a mechanism as primitive. Although a function can remove an element of the state and then place it back unchanged, we demonstrate in Chapter 3 that such a programming trick is not appropriate when multiple functions are being executed in parallel and wish to examine the same elements of the state.

Aborting and deadlocking primitive functions

MSTLs such as Γ and CGP lack a notion of primitive functions which abort or deadlock. When used as coordination languages for programs which *can* exhibit deadlocking and abortion, they cannot but be inappropriate. For example, we may wish to coordinate a number of badly-written C programs using Γ. The chances are that one of these programs will abort with a core dump. The semantics of the coordination language has to be able to cope with this gracefully.

We show how primitive functions exhibiting aborting and/or deadlocking primitive functions can be incorporated into an MSTL in Chapter 2.

Unstructured data hides sequentiality

MST enthusiasts frequently claim that the multisets' unstructured nature makes for highly parallel implementations [16, 20, 65]. Although this *may* be the case, it need not be the case. Many programs appear to be highly parallel, because the programs contain a large amount of data or a large number of functions composed in parallel. However, the multiset's lack of structure hides the real data dependencies, which in practice will severely limit the

available parallelism. Consider the following Γ program, which has an infinite number of functions in a parallel composition, but a maximal parallelism of one. We use Pnm to indicate a primitive function $(\lambda x.x = n, \lambda x.\{m\})$ (i.e. a primitive function that converts an n into an m, if an n is found in the multiset).

$$\langle P01 + P12 + P23 + \ldots, \{0\}\rangle$$

At the other extreme, we can write Γ programs containing no interleaving or parallel compositions, but which can be executed with an infinite parallelism:

$$\begin{array}{l}\langle (R, A), \{0, 1, 2, \ldots\}\rangle \\ \text{where} \\ \qquad \begin{array}{lcl} R(x) & = & \text{True} \\ A(x) & = & \{x + 1\} \end{array}\end{array}$$

Aside from these two pathological examples, we frequently find that an apparently parallel program is afflicted by hidden data dependencies. The following example is a transparent case:

$$P12 + P21$$

The potential parallelism of this program depends upon the values in the multiset. Therefore, techniques to derive the program's parallelism must examine the multiset. Unfortunately, the multiset is often not available to a compiler analysis, being provided later as input to a program.

The problem is that logical parallelism is a property of a algorithm while physical parallelism is a property of an implementation. Converting the former into the latter is not easy [2]. Real understanding of the parallelism available to a particular MST program and the extraction of that parallelism by a compiler, remain non-trivial tasks. So even in the case of MSTLs, there appears to be no such thing as free parallelism.

1.3.4 Γ's problems

In the previous section, we concentrated on problems common to all MSTLs. This (smaller) section draws the readers attention to a few of Γ's more obscure difficulties. Since the research reported in this thesis stands on Γ's shoulders, much of the research has been driven by Γ's idiosyncrasies.

One-shot functions

Γ lacks one-shot functions. That is, even a primitive function in Γ will continue to be applied to the multiset until it can be applied no longer. As Sands has observed [126, 127], this behaviour implies that no 'big bang' program exists. That is, there is no Γ program which can be successfully applied to the empty multiset once and then terminates. This behaviour is unfortunate, because without one-shot functions we can't even initialise

our programs. A good example of this is when a user is required to input some information which is used to generate the initial multiset. Without the possibility of writing proper initialisation routines, this kind of program becomes almost impossible to write in practice, as the programmer fights to force termination of the initialisation functions and to let the program proper start.

Recursion and conditionals

Γ lacks a general notion of recursion. In other words, although recursion exists in Γ—all primitive functions are implicitly recursive—it is not possible to write arbitrary Γ programs which are recursive. For example, consider a program which asks the user to input a list and then sorts it. We can write the program like this:

$$\langle sort\_list \circ get\_list\_from\_user, \emptyset \rangle$$

But now consider that we wish to write a program to *repeatedly* perform this task. In conventional Γ, we have no recursive operators which are applicable to programs, so our example is almost impossible to write (see Chapter 5 for an example of this).

This problem has been addressed in [65], wherein a solution was offered in the form of a binary iterator operator '$*$'. $P * Q$ repeatedly applies P then Q until neither can be applied. The iterator therefore allows us to write the above program. Unfortunately, adding the iterator does not solve the problem of one-shot functions, mentioned above.

As an alternative mechanism, we propose that all primitive functions are one-shot. That is, regardless of its success, a primitive function will only be applied once. To build recursive programs, we suggest that the programmer make use of conventional recursive constructs (such as 'μ-operators, used in 2). In order to ensure termination when a program is *not* applicable as well as recursion when it *is*, we also require the presence of a conditional in the language. Our proposal is introduced and examined in Chapter 2.

1.4 The motivation for this thesis

This thesis discusses aspects of multiset transformation languages from the point of view of the applications. While there is still a certain amount of formalism in the thesis (about half the thesis is formal semantics), we allow practical considerations to drive the research to the extent of motivating the addition of new capabilities to the formalisms. In particular, our PT formalism (introduced in Chapter 2) is a product of attempting to write the applications of Chapters 4 and 5.

Our choice of orientation motivates us to propose a new multiset transformation model and describe a number of applications. Most of these applications come from theoretical biology, where a large amount of work has been done in modelling plant growth using formal languages (see Chapter 3 for more details).

1.4.1 The structure of this thesis

Broadly speaking, the thesis contains two parts. The first, consisting of Chapters 2 and 3, is a formal treatment of control-flow coordination in MSTLs. In these chapters, we develop a unified framework for parallel transformation, called Parallel Transformation (PT)[8]. We show that PT can be used to encode many formalisms for parallel programming. By presenting a unified framework, we make it possible to describe a great variety of different behaviours for programming constructs. This, in turn, enables us to compare and contrast the particular advantages and disadvantages of each construct. In Chapter 2, the discussion concentrates on issues arising when programs exist in interleaving compositions with other programs. Constructs which behave differently in different program-contexts we refer to as *program context-sensitive*. In addition, we discuss interleaving operators which synchronise when none of their components can be successfully applied to the multiset. We call such operators *failure synchronous*. The chapter is abundantly illustrated with examples.

In Chapter 3, we discuss the influence on program reduction of the ability to read elements from the multiset without removing them. Such elements we christen the *data-context* of the reduction. We also consider the effect of reductions which can happen logically simultaneously: true parallelism. Operators which can perform in this way we call *success-synchronous* as an antonym to the failure synchrony presented in the previous chapter. Examples and motivations are given throughout.

The second part of the thesis is more application-oriented and consists of Chapters 4 and 5. All of the examples given are small, although we hope to convince the reader of the applicability of MSTLs to a wide variety of application areas. In particular, our examples demonstrate MSTLs used to coordinate very simple primitive functions. The most important class of such programs is that of production systems [91]. In real-world applications, we would expect that coordinated components would be complete software systems in their own right. Unfortunately, such systems are not examined in this thesis: it was decided to work towards a full understanding of the basic concepts involved in coordination before undertaking any large-scale projects.

In Chapter 4, we discuss how to make multiset rewrites in MSTLs probabilistic, so that stochastic applications such as those prevalent in the physical modelling community, can be modelled. As ever, the chapter is illustrated with a posse of examples. The second of the two chapters, Chapter 5, describes applications of MSTLs to biological modelling. We demonstrate a number of fractal plant-like objects and a cellular automaton. Our examples demonstrate how the ability of MSTLs to coordinate several primitive functions pays rich dividends when modelling this kind of system.

After the four chapters of 'meat', we conclude the thesis with the dessert: a summing up, a drawing of conclusions and a discussion of possible future work.

[8] One referee has chastised me for this 'uninformative' name. To him, I suggest that PT stands for 'Pointless Terminology'.

Chapter 2

A unified semantic framework for parallel transformation[1]

In this chapter, we offer PT, a parameterised structural operational semantics (SOS) which abstracts from interaction with the state and from the dynamics of program reduction. The SOS can be used to mimic particular languages by choosing the appropriate parameters. Interaction with the state is determined by parameters which give both the form of the state and describe the result of applying a primitive function to the state. The dynamics of reduction are controlled through quantifying how *interesting* a particular reduction is, where the most interesting offered reduction is chosen when more than one reduction is possible. Furthermore, the parameters control what happens when none of the possible reductions is 'interesting enough', thereby capturing a number of synchronisation effects. PT is therefore a flexible semantics which can be used as a tool for language classification, for language design and for comparing different dialects of the same language.

We use PT to analyse the program context-sensitivity and synchronisation on failure properties of a number of multiset transformation languages (we restrict ourselves to multiset transformation languages for reasons of space). Some of the languages here compared are new, while some have already been proposed in the literature, such as Gamma (Γ) and Calculus of Gamma Programs (CGP). All of the languages are translated into PT to facilitate comparison of their (dis-) similarities. The result of our comparison is a taxonomy of the languages, based on their parameters. Our classification is constructed in such a way that it is possible to define new languages to occupy the currently blank areas of the taxonomy.

[1]This chapter is an extended version of the paper "Context sensitivity and synchronisation as taxonomics for parallel programming" which appeared in the 30th Hawaii International Conference on System Sciences, Maui, Hawaii in January 1997 [96].

2.1 Introduction

Many languages for parallel programming include a binary operator indicating that reductions of its two components are to be interleaved. The question arises of *how* they are to be interleaved. Aside from questions of fairness [4], we could ask whether or not the choice of which component to reduce next is dependent upon the possible transformations which both components can make. For example, we can imagine wanting component A of the composition $A \,|||\, B$ to be chosen every time that A can be successfully applied reduce and B cannot. Interleaving operators which behave in this way possess the *program context-sensitive reduction property* (in the terminology of [37]). For example, the interleaving operators of Ciancarini's CGP [37] and of Gamma [20] are program context-sensitive (CS), but in different ways. In contrast, program context-*in*sensitive (CI) operators simply choose one component of the composition and reduce it by one step, without regard to those reductions (if any) offered by the other component of the composition. The interleaving operators of `while`-like languages tend to be context-insensitive [28]. We should distinguish CS versus CI as described in this chapter from internal and external choices as described in [109]. Internal choices are not interleaving choices and are always CI while external choices *are* interleaving choices and may be either CS or CI.

In addition to being CS or CI, some interleaving operators reduce both of their components synchronously when neither can perform any reductions. We call languages with this property *failure synchronous* (FS). Gamma, CGP and Dijkstra's Guarded Command Language (GCL) [46] are FS, but in different ways. Other interleaving operators (such as those in conventional `while` languages), are not FS. Notice that failure synchrony is only an interesting property for a language to possess when that language is CS. In CI languages, the notion of some reductions being favoured over others does not arise: all reductions are *a priori* equally interesting. Furthermore, interleaving compositions of programs can behave in different ways if one of them aborts or deadlocks.

Given the plethora of different varieties of interleaving, it would be useful to have a framework within which several (or, at best, all) of the different flavours could be described. This would allow a formal comparison of the possibilities each offers. In particular, such a framework would allow us to investigate the effects of incorporating different interleaving operators in the same language and to appreciate the differences made to a language by a particular operator.

In this chapter, we attempt to give such a unifying framework, which we christen PT (for *P*arallel *T*ransformation). Fundamental to PT is a formalisation of how *interesting* a reduction is, such that the most interesting reduction is chosen, when more than one reduction is possible. By specifying the levels of interest associated with every possible reduction, we can control which reductions must be chosen in which situations.

In Section 2.2, we describe PT both informally and formally. In Section 2.3, we give translations into PT of a number of multiset transformation

languages including Γ, CGP and a number of new formalisms. In Sections 2.4 and 2.5, we use PT as a vehicle for discussing the differences between a number of interleaving operators. Finally, we mention some related work, discuss possible avenues for future exploration and draw our conclusions.

2.2 Introducing PT

PT consists of a parameterisable structural operational semantics (SOS [114]) which abstracts from both the state and the level of interest associated with each reduction. We choose an SOS as our vehicle both because (i) all the languages which we here encode into PT have been presented in terms of an SOS and (ii) because we are interested in an *operational* view of different interleaving operators. Our latter motivation stems from our desire to use a semantics as the basis for an implementation, with which real programs can be executed.

Our reasons for parameterising the interaction with the state and the levels of interest associated with each reduction, are twofold:

- We want to be able to encode a number of variants of interleaving composition within PT. To do this, we introduce the notion of levels of interest.
- We want to be able to encode, in PT, a number of formalisms which make use of different kinds of state (functions, multisets, sets of variable-value pairs etc). For this purpose, we abstract from the state of the computation.

Abstraction from the state and from the levels of interest are performed by introducing the notions of a *selection* function ($\mathcal{S}$ function) and an *interest* tuple (i-tuple). An $\mathcal{S}$ function provides an interface between the program and the state. By defining the interface between the program and the state, it enforces the form of the state and enforces what constitutes a stable state. We shall see the exact rôle of the $\mathcal{S}$ function and the i-tuple in the formal semantics of PT, given below. Stability of a state is important because recursive (sub-) programs terminate when a fixed point is reached i.e. when the state is stable with respect to that program. Thus, the precise definition of stability defined in the $\mathcal{S}$ function, determines the termination condition for (recursive) programs. Primitive functions (described in Section 2.2.1) use $\mathcal{S}$ functions to mediate their interaction with the state. A certain amount of computation (such as a state update) is associated with an $\mathcal{S}$ function.

An i-tuple is a four-tuple containing the levels of interest associated with certain kinds of reduction. It is these levels of interest which are used to determine which offered reductions are most interesting and therefore which reductions can be performed. We explain both $\mathcal{S}$ functions and i-tuples formally in Section 2.2.2.

A particular parallel language is defined by giving an i-tuple, a $\mathcal{S}$ function for every possible kind of interaction between the program and the state and a set of construct definitions in terms of the connectives available in PT.

PT therefore defines a family of languages, at least one for each i-tuple and combination of $\mathcal{S}$ functions.

2.2.1 The syntax of PT

We introduce the following syntactic categories for PT:

d	$\in$	$\mathbf{D}$	$\stackrel{\text{def}}{=}$	(data)	
σ	$\in$	Σ	$\stackrel{\text{def}}{=}$	$(\wp\mathbf{D})^* \cup \{\bot, \Delta\}$	finite states
B	$\in$	$\mathbf{B}$	$\stackrel{\text{def}}{=}$	$\mathbf{D}^* \to \mathbf{Bool}$	predicates
A	$\in$	$\mathbf{A}$	$\stackrel{\text{def}}{=}$	$\mathbf{D}^* \to \Sigma$	action functions
α, β	$\in$	$\mathbf{L}$	$\stackrel{\text{def}}{=}$	$(\mathbb{N}_\mathbf{0}, \leq)$	labels
$\mathcal{O}$	$\in$	$\mathbf{O}$	$\stackrel{\text{def}}{=}$	$\mathbf{L} \times \mathbf{L} \times \Sigma$	$\mathcal{S}$ function output
$\mathcal{S}$	$\in$	$\mathbf{S}$	$\stackrel{\text{def}}{=}$	$\mathbf{B} \to \mathbf{A} \to \Sigma \to \mathbf{O}$	$\mathcal{S}$ functions
i	$\in$	$\mathbf{I}$	$\stackrel{\text{def}}{=}$	$\mathbf{L} \times \mathbf{L} \times \mathbf{L} \times \mathbf{L}$	i-tuples

We leave the underlying domain $\mathbf{D}$ undefined, although it can be assumed, for the sake of intuition, to include at least integers, booleans and tuples thereof. $\wp\mathbf{D}$ is the power multiset of $\mathbf{D}$. $\bot$ is used to indicate abortion. Δ is deadlock. $\mathbf{Bool} = \{$ True, False $\}$. $\mathbb{N}_0$ is the set of natural numbers, starting at zero. $(\mathbb{N}_0, \leq)$ is a reflexively totally-ordered set. We order the labels so that the notion of 'largest label' is well-defined. The labels offered by a reduction are used to calculate which possible reductions are most interesting.

$P \in \mathbf{P}$	$\stackrel{\text{def}}{=}$	ϵ	the empty program
		X	program variable
		$(B, A, \mathcal{S})$	primitive function
		$P_1 \fatsemi P_2$	sequence
		$P_1 \mid P_2$	choice
		$P_1 \mid\mid\mid P_2$	interleave
		$P_1 \stackrel{\mathbf{n}}{\rhd} P_2 : P_3$	condition
		$\mu X.P$	recursion

Figure 2.1: The abstract syntax of PT.

The abstract syntax for PT is given in Fig. 2.1. We give the intuitive meanings of the various operators. All of these behaviours are described formally in Section 2.2.3. A primitive function is applied to the state and is then discarded. Primitive functions are the only functions which can either read or alter the state. $P \fatsemi Q$ indicates that P is to be reduced before Q. $P \mid\mid\mid Q$ is an interleaving composition. $P \mid Q$ is a non-deterministic choice. A recursive program is written $\mu X.P$, where X may be free in P.

The conditional is non-standard. $P \overset{\mathbf{n}}{\triangleright} Q : R$ can be read as 'if P then Q else R', where the reduction of program P can change the state of the computation. The purpose of $\mathbf{n}$ in $P \overset{\mathbf{n}}{\triangleright} Q : R$ is to keep track of the maximum level of interest associated with any reduction of P. We shall see in the SOS of PT that, if the (maximum) level of interest associated with a reduction is greater than a certain threshold, then $\epsilon \overset{\mathbf{n}}{\triangleright} Q : R$ will choose Q; otherwise it chooses R. The conditional therefore corresponds to a more general version of a conventional if ... then ... else ... conditional, where the condition is an arbitrary program which is applied to the multiset instead of (as is usually the case) a truth-valued function. The use here therefore corresponds to that of $P \ ? \ Q : R$ in C [82].

2.2.2 The anatomy of a selection function and of an interest tuple

We have already said that we wish our interleaving operator to pick, at every step, the most interesting possible reduction. In order to do this, every reduction has a pair of labels associated. with it. As we (formally) stated in Section 2.2.1, these labels are natural numbers. For each derivation step, the labels of all possible reductions are calculated on the basis of which transitions each (sub-) program can perform.

The labels originate with the selection function, which defines how a primitive function is applied to the state (we ignore the case here of rewriting of empty programs). The labels then percolate down through an SOS proof tree (perhaps being modified along the way) until they get to the root. In the root of the tree, the actual reduction to be performed is chosen: it is simply the most interesting one offered. The key to understanding the levels of interest associated with PT reductions is therefore an understanding of the labels returned by the selection function and the ways in which these affect the final reduction choice taken.

We give an example $\mathcal{S}$ function and i-tuple and explain their components. The type of an $\mathcal{S}$ function is given in section 2.2.1.

$$\mathcal{S}_\Gamma(B,A,\sigma) \quad \overset{\text{def}}{=} \begin{cases} (\texttt{succ}, \texttt{succ}, \sigma[A\vec{\sigma}/\vec{\sigma}]) & \text{if } \exists \vec{\sigma} \subseteq \sigma . B\vec{\sigma} \\ (\texttt{fail}, \texttt{fail}, \sigma) & \text{otherwise} \end{cases}$$

$$i_\Gamma \quad \overset{\text{def}}{=} \quad (\texttt{semi}, \texttt{cond}, \texttt{sync}, \texttt{if})$$

where $\quad \mathbf{L} \ni \texttt{semi}, \texttt{cond}, \texttt{sync}, \texttt{if}, \texttt{succ}, \texttt{fail}$

and $\quad (\texttt{if} = \texttt{fail}) < \texttt{sync} \leq (\texttt{semi} = \texttt{cond} = \texttt{succ})$

The selection function $\mathcal{S}_\Gamma$ given above applies a condition B to a multiset σ. On the right-hand side of the function, we use some shorthand to explain how the condition is applied to the multiset. The condition is applied by searching for a subset of the multiset which, when ordered in some way, causes condition B to be true. The ordered sub-multiset of σ is written $\vec{\sigma}$. If there *is* such a subset of the multiset, then the left-hand side of the selection

function indicates that the elements of $\vec{\sigma}$ in the multiset are replaced by $A\vec{\sigma}$. Labels `succ` and `succ` are also returned, indicating that the primitive function was successfully applied to the multiset. If no such sequence of elements can be found, then the old state is returned, together with the labels `fail` and `fail`, which indicate that the attempted primitive function application failed. We shall see this description of the use of the selection function formalised in Section 2.2.3.

Observing the syntax of PT in Fig. 2.1, we notice that the program composition operators are sequential composition, interleaving composition and the conditional. All three of these make use of or alter the levels of interest associated with reductions of their components. The latter two make use of the levels of interest associated with their components to determine which sub-program to choose. For interleaving composition, a non-deterministic choice is made between two possible reductions. The conditional makes a deterministic choice between the two possible right-hand sides of the operator (the 'then' and the 'else' parts). Of all the operators in PT, the behaviour of only these two are affected by the levels of interest associated with a reduction. To capture this effect, performing a reduction yields two labels, the first of which indicates how interesting the conditional finds the reduction ('the level of interest from the point of view of the conditional') and the second of which indicates how interesting the interleaving operator finds the reduction ('the level of interest from the point of view of interleaving'). The presence of a reflexive total order on the labels means that some labels are greater than others and, therefore, indicate more interesting reductions. The ordering on the labels are given above. For each i-tuple and, therefore, each class of languages, a different ordering on the labels can be given. The potential for specifying different orderings on the labels gives rise to much of the power of PT (the rest comes from the ability to specify different selection functions for every primitive function). The levels of interest determine the actual choice of reduction taken in two ways:

- The interleaving operator chooses the most interesting reduction which it can perform, if that reduction is at least as interesting as `sync`. If it is not, then one or both of the components of the interleaving operator are rewritten without altering the state.

- The conditional rewrites to its 'then' branch or its 'else' branch. Which one of these transitions occurs depends upon how interesting were the reductions performed on the condition. The level of interest which the conditional considers 'interesting enough' to rewrite to its 'then' clause is specified by `cond`.

Parameter `semi` indicates the minimum level of interest associated with a sequential composition, from the point of view of interleaving composition. In other words, any reduction within a sequential composition has a level of interest for the interleaving composition of at least `semi`. Parameter `sync` gives the level of interest required before a transition in an interleaving composition is considered interesting enough to perform. If both reductions

available to an interleaving composition are smaller than `sync`, then one or both components of the composition are rewritten without altering the state. Parameter `cond` gives the level of interest required for the condition $\epsilon \overset{\mathbf{n}}{\triangleright} Q : R$ to rewrite to Q. If $\mathbf{n} < \texttt{cond}$, then $\epsilon \overset{\mathbf{n}}{\triangleright} Q : R$ is interesting enough to rewrite to R. Finally, label `if` gives the minimal level of interest associated with a conditional, from the point of view of interleaving composition.

2.2.3 The formal semantics of PT

We introduce configurations:

$$\langle P, \sigma \rangle \quad \in \quad \mathbf{Conf} \quad \stackrel{\text{def}}{=} \quad \mathbf{P} \times \Sigma \quad \text{configurations}$$

We can now give an operational semantics to PT using the labelled transition system $(\mathbf{P}, \mathbf{L}, \{\overset{(\alpha,\beta)}{\Longrightarrow} | \ \alpha, \beta \in \mathbf{L}\})$. The labels α and β are used to convey the levels of interest associated with a reduction from, respectively, the points of view of the conditional and the interleaving operator. The SOS of PT is given in Fig. 2.2. A short explanation of the rules follows.

The function application rules $\mathbf{App_1}$ and $\mathbf{App_2}$ apply a primitive function to the state, yielding a new state and a pair of labels. The labels are generated by the selection function and indicate the levels of interest associated with the (attempted) function application. The meaning of $\exists \vec{\sigma} \subseteq \sigma$ in $\mathbf{App_1}$ and $\mathbf{App_2}$ is explained on Page 13. If the applied primitive function does not deadlock, then $\mathbf{App_1}$ throws the primitive function away (reduces it to the empty program) and updates the state. If the applied primitive function *does* deadlock, then $\mathbf{App_2}$ ensures that the deadlocking program is retained. $\mathbf{Chs_L}$ and $\mathbf{Chs_R}$ make a non-deterministic choice between, respectively, their left or right components. This choice is *internal* in the sense of [69]. $\mathbf{Int_1}$ throws away the resulting program and state of untaken transitions, but requires the labels to determine which of the transitions is most interesting. Furthermore, $\mathbf{Int_2}$ states that, if neither reduction is at least as interesting as `sync`, but one reduction is nevertheless more interesting than the other, the program associated with the most interesting reduction is reduced without altering the state. $\mathbf{Int_3}$ states that, if both reductions are equally interesting and are less interesting than `sync`, both resulting states are thrown away and both programs are synchronously reduced. $\mathbf{Int_2}$ and $\mathbf{Int_3}$ allow us to capture failure synchronous operators, wherein one or both inapplicable functions are reduced without affecting the state. The $\mathbf{Int}_{\epsilon_\mathbf{L}}$ and $\mathbf{Int}_{\epsilon_\mathbf{R}}$ rules remove empty programs from interleaving compositions of programs.

It is important to understand exactly what is happening in the rules for reducing interleaved compositions of programs. Since we are working with an SOS, every actual reduction taken by a program (fragment) is the root of a tree generated by applying the SOS rules to the configuration before the reduction takes place. The root of the generated tree is the reduction which is actually taken. In order to determine which reduction will actually be performed, the tree also includes 'explorations' of reductions which are

not actually taken (i.e. the less interesting reductions). It does this in order to determine what is the actual level of interest associated with those reductions. Therefore, we have a semantics wherein *all* components of an interleaving composition are conceptually applied to the multiset to see how interesting each reduction is and what the resulting configuration after each reduction is. Once we know which of the possible reductions is the most interesting, we keep the configuration resulting from that, most interesting, reduction and discard the results of all other generated reductions.

In terms of semantics, this blatant inefficiency is not a problem: we are simply exploring a space to see what we wish to do. In terms of an actual implementation of a language, however, applying all interleaved programs to the state and then throwing most of the results away is clearly a waste of cycles. We would hope that compiler builders would address this issue. On the other hand, the inefficiency is trivial as compared to the inefficiency of searching for data in multisets in the first place, a point which we made in 1.3.1 in connection with reducing the number of programs in an interleaving composition.

The sequential composition rules **Seq** and $\mathbf{Seq}_\epsilon$ are standard, except that the label of the consequent of **Seq** is calculated from the label of the antecedent using $\sqcup$, which is the max function on labels (natural numbers). The conditional rule **If** proceeds by applying the condition to the state and using the results of that evaluation to determine how interesting the reduction is from the point of view of the condition itself. Rule **If**, like rule **Seq**, calculates a label of its consequent using the max function. Rules $\mathbf{If}_{\epsilon_1}$ and $\mathbf{If}_{\epsilon_2}$ reduce conditionals with empty conditions into one of either two programs, depending on whether any previous reduction of the condition was at least as interesting as `cond`. The rule for recursion (**Rec**) is also standard, involving an unfolding of the recursive program [114]. It states that all the free instances of X in P are replaced by $\mu X.P$. For the sake of brevity, we sometimes write P^* for $\mu X.P \overset{\mathbf{0}}{\triangleright} X : \epsilon$ and $P \overset{\mathbf{n}}{\triangleright} Q$ for $P \overset{\mathbf{n}}{\triangleright} Q : \epsilon$.

If the current state is $\perp$ or Δ, only reductions which remove empty programs are performed. This behaviour is evidenced by the presence of $\sigma \notin \{\perp, \Delta\}$ in many of the rules of the SOS.

Rules **If** and **Seq** calculate the max function on a label and a parameter (a member of the i-tuple). The intention is that the approach makes it possible for reduction of a composed program to be more interesting than reduction of its left-hand side. Such a treatment makes it possible to capture formalisms wherein reduction of a program in a composition may be more interesting than reduction of the same program outside a composition. Such behaviour is typical of languages such as Γ and CGP, as we shall see.

2.3 A posse of parallel languages

With our formal machinery in place, we can demonstrate the flexibility of PT by giving a number of translations of parallel languages into PT. In each translation, the properties of the reduction relation for that language are

$$\frac{\mathcal{S}(B,A,\sigma) = (\mathbf{c_1},\mathbf{c_2},\sigma')}{\langle (B,A,\mathcal{S}),\sigma\rangle \overset{(\mathbf{c_1},\mathbf{c_2})}{\Longrightarrow} \langle \epsilon,\sigma'\rangle}\ \mathbf{App_1} \quad \text{if } \sigma \notin \{\bot,\Delta\} \wedge \sigma' \neq \Delta$$

$$\frac{\mathcal{S}(B,A,\sigma) = (\mathbf{c_1},\mathbf{c_2},\Delta)}{\langle (B,A,\mathcal{S}),\sigma\rangle \overset{(\mathbf{c_1},\mathbf{c_2})}{\Longrightarrow} \langle (B,A,\mathcal{S}),\Delta\rangle}\ \mathbf{App_2} \quad \text{if } \sigma \notin \{\bot,\Delta\}$$

$$\langle P \mid Q,\sigma\rangle \overset{(\mathbf{0},\infty)}{\Longrightarrow} \langle P,\sigma\rangle\ \mathbf{Chs_L} \quad \text{if } \sigma \notin \{\bot,\Delta\}$$

$$\langle P \mid Q,\sigma\rangle \overset{(\mathbf{0},\infty)}{\Longrightarrow} \langle Q,\sigma\rangle\ \mathbf{Chs_R} \quad \text{if } \sigma \notin \{\bot,\Delta\}$$

$$\frac{\langle P_1,\sigma\rangle \overset{(\mathbf{c_1},\mathbf{c_2})}{\Longrightarrow} \langle P_1',\sigma'\rangle \quad \langle P_2,\sigma\rangle \overset{(\mathbf{d_1},\mathbf{d_2})}{\Longrightarrow} \langle P_2',\sigma''\rangle}{\langle P_1 \,|||\, P_2,\sigma\rangle \overset{(\mathbf{c_1},\mathbf{c_2})}{\Longrightarrow} \langle P_1' \,|||\, P_2,\sigma'\rangle \,\&\, \langle P_2 \,|||\, P_1,\sigma\rangle \overset{(\mathbf{c_1},\mathbf{c_2})}{\Longrightarrow} \langle P_2 \,|||\, P_1',\sigma'\rangle}\ \mathbf{Int_1}$$
$$\text{if } d_2 \leq c_2 \wedge \mathtt{sync} \leq c_2 \wedge \sigma \notin \{\bot,\Delta\}$$

$$\frac{\langle P_1,\sigma\rangle \overset{(\mathbf{c_1},\mathbf{c_2})}{\Longrightarrow} \langle P_1',\sigma'\rangle \quad \langle P_2,\sigma\rangle \overset{(\mathbf{d_1},\mathbf{d_2})}{\Longrightarrow} \langle P_2',\sigma''\rangle}{\langle P_1 \,|||\, P_2,\sigma\rangle \overset{(\mathbf{c_1},\mathbf{c_2})}{\Longrightarrow} \langle P_1' \,|||\, P_2,\sigma\rangle \,\&\, \langle P_2 \,|||\, P_1,\sigma\rangle \overset{(\mathbf{c_1},\mathbf{c_2})}{\Longrightarrow} \langle P_2 \,|||\, P_1',\sigma\rangle}\ \mathbf{Int_2}$$
$$\text{if } d_2 < c_2 < \mathtt{sync} \wedge \sigma \notin \{\bot,\Delta\}$$

$$\frac{\langle P_1,\sigma\rangle \overset{(\mathbf{c_1},\mathbf{c_2})}{\Longrightarrow} \langle P_1',\sigma'\rangle \quad \langle P_2,\sigma\rangle \overset{(\mathbf{d_1},\mathbf{d_2})}{\Longrightarrow} \langle P_2',\sigma''\rangle}{\langle P_1 \,|||\, P_2,\sigma\rangle \overset{(\mathbf{c_1}\sqcup\mathbf{d_1},\mathbf{c_2})}{\Longrightarrow} \langle P_1' \,|||\, P_2',\sigma\rangle}\ \mathbf{Int_3}$$
$$\text{if } (c_2 = d_2) < \mathtt{sync} \wedge \sigma \notin \{\bot,\Delta\}$$

$$\langle P_1 \,|||\, \epsilon,\sigma\rangle \overset{(\mathbf{0},\infty)}{\Longrightarrow} \langle P_1,\sigma\rangle\ \mathbf{Int}_{\epsilon_\mathbf{L}} \qquad \langle \epsilon \,|||\, P_2,\sigma\rangle \overset{(\mathbf{0},\infty)}{\Longrightarrow} \langle P_2,\sigma\rangle\ \mathbf{Int}_{\epsilon_\mathbf{R}}$$

$$\frac{\langle P_1,\sigma\rangle \overset{(\mathbf{c_1},\mathbf{c_2})}{\Longrightarrow} \langle P_1',\sigma'\rangle}{\langle P_1 ; P_2,\sigma\rangle \overset{(\mathbf{c_1},(\mathbf{c_2}\sqcup\mathtt{semi}))}{\Longrightarrow} \langle P_1' ; P_2,\sigma'\rangle}\ \mathbf{Seq} \quad \text{if } \sigma \notin \{\bot,\Delta\} \qquad \langle \epsilon ; P_2,\sigma\rangle \overset{(\mathbf{0},\infty)}{\Longrightarrow} \langle P_2,\sigma\rangle\ \mathbf{Seq}_\epsilon$$

$$\frac{\langle P_1,\sigma\rangle \overset{(\mathbf{c_1},\mathbf{c_2})}{\Longrightarrow} \langle P_1',\sigma'\rangle}{\langle P_1 \overset{\mathbf{n}}{\triangleright} P_2 : P_3,\sigma\rangle \overset{(\mathbf{c_1},(\mathbf{c_2}\sqcup\mathtt{if}))}{\Longrightarrow} \langle P_1' \overset{\mathbf{n}\sqcup\mathbf{c_1}}{\triangleright} P_2 : P_3,\sigma'\rangle}\ \mathbf{If} \quad \text{if } \sigma \notin \{\bot,\Delta\}$$

$$\langle \epsilon \overset{\mathbf{n}}{\triangleright} P_2 : P_3,\sigma\rangle \overset{(\mathbf{0},\infty)}{\Longrightarrow} \langle P_2,\sigma\rangle\ \mathbf{If}_{\epsilon_1} \quad \text{if } n \geq \mathtt{cond} \qquad \langle \epsilon \overset{\mathbf{n}}{\triangleright} P_2 : P_3,\sigma\rangle \overset{(\mathbf{0},\infty)}{\Longrightarrow} \langle P_3,\sigma\rangle\ \mathbf{If}_{\epsilon_2} \quad \text{if } n < \mathtt{cond}$$

$$\langle \mu X.P,\sigma\rangle \overset{(\mathbf{0},\infty)}{\Longrightarrow} \langle P[\mu X.P/X],\sigma\rangle\ \mathbf{Rec} \quad \text{if } \sigma \notin \{\bot,\Delta\}$$

Figure 2.2: The SOS of PT. The family of one-step transition relations $\overset{(\mathbf{c_1},\mathbf{c_2})}{\Longrightarrow}$, on configurations is the least satisfying the above clauses. Notice that $\mathcal{S}$ and $i = (\mathtt{semi},\mathtt{cond},\mathtt{sync},\mathtt{if})$ are unspecified. $\mathbf{Int_2}$ and $\mathbf{Int_3}$ have two consequents apiece. They are non-deterministic rules which can choose either consequent if both are applicable.

captured through an appropriate choice of selection functions, i-tuples and encodings into the connectives of PT. More examples of the flexibility of PT can be seen in Chapter 3.

2.3.1 Γ

Γ [20] was proposed as a model of parallel programming in which programs can be written which contain a minimum of explicit control flow. It is described in Chapter 1. The SOS of Γ is given there and in [65]. A possible translation of Γ into PT is given below. In Section 2.5, we will see that at least one other (equivalent) translation is possible.

$$
\begin{array}{lll}
i_\Gamma & \stackrel{\text{def}}{=} & (\mathtt{semi}, \mathtt{cond}, \mathtt{sync}, \mathtt{if}) \\
\mathcal{S}_\Gamma(B,A,\sigma) & \stackrel{\text{def}}{=} & \begin{cases} (\mathtt{succ}, \mathtt{succ}, \sigma[A\vec{\sigma}/\vec{\sigma}]) & \text{if } \exists \vec{\sigma} \subseteq \sigma . B\vec{\sigma} \\ (\mathtt{fail}, \mathtt{fail}, \sigma) & \text{otherwise} \end{cases}
\end{array}
$$

where:
$(\mathbf{0} = \mathtt{if} = \mathtt{fail}) < \mathtt{sync} \leq (\mathtt{semi} = \mathtt{cond} = \mathtt{succ}) < \infty$

$$
\begin{array}{lcl}
[\![(B,A)_\Gamma]\!] & = & (B, A, \mathcal{S}_\Gamma)^* \\
[\![P_2 \circ_\Gamma P_1]\!] & = & [\![P_1]\!] \fatsemi [\![P_2]\!] \\
[\![P_1 +_\Gamma P_2]\!] & = & [\![P_1]\!] \,|||\, [\![P_2]\!]
\end{array}
$$

We can satisfy our intuitions that this translation behaves in the same way as Γ through examination of the parameters. Examining the translation, we see that $\mathtt{fail} < (\mathtt{semi} = \mathtt{succ})$. Therefore, reduction of the left-hand side of a sequential composition $P \fatsemi Q$(which has a level of interest of at least $\mathtt{semi}$) is as likely to be chosen, when it is offered, as successful application of a primitive function (level of interest: $\mathtt{succ}$). A failed application of a primitive function is the least interesting kind of reduction and can therefore only be picked when all functions are inapplicable primitive functions. Therefore, *only* if neither an applicable primitive function nor a sequential composition is offered can an inapplicable primitive function be discarded. Such a situation occurs only when the program contains nothing but inapplicable primitive functions. In such a case, all of the primitive functions are thrown away synchronously. Those familiar with Γ will recognise this behaviour. In Section 2.4 and 2.5, we give example executions of Γ programs, which highlight the curious properties of its interleaving operator.

Proposition 1 (Correctness of the translation) *Our translation of* Γ *into PT is correct.*

Proof by induction over the structure of the transition relation. The proof is to be found in Appendix A.1.

2.3.2 Γ^μ: Γ with loops and single-shot functions

As we said in the introduction, one of the weaknesses of the Γ model arises because of the restricted way in which recursion is made available. For

example, many of the applications which we have described in Chapters 4 and 5 require a more sophisticated treatment of recursion and single-shot functions than those offered in Γ, in order to make feasible, applications programming from a software engineering point of view. A good example of this is an initialisation: the state is initialised once in the course of a program's execution. The initialising function should therefore be applied exactly once. One-shot functions are not available in Γ, although they can sometimes be simulated by clever use of tagging (but not always: see [127]).

PT offers us general recursion and primitive functions which are one-shot. Using PT as a guide, we can extend Γ so that arbitrary recursive programs can be written. We define a new language Γ^{μ}, so-called because it is related to Γ, but makes recursion (the μ-operator) explicit.

$$
\begin{array}{lcl}
i_{\Gamma^{\mu}} & \stackrel{\text{def}}{=} & i_{\Gamma} \\
[\![(B,A)_{\Gamma^{\mu}}]\!] & \stackrel{\text{def}}{=} & (B,A,\mathcal{S}_{\Gamma}) \\
[\![P_2 \circ_{\Gamma^{\mu}} P_1]\!] & \stackrel{\text{def}}{=} & [\![P_1]\!] \fatsemi [\![P_2]\!] \\
[\![P_1 +_{\Gamma^{\mu}} P_2]\!] & \stackrel{\text{def}}{=} & [\![P_1]\!] \,|||\, [\![P_2]\!] \\
[\![P^{*_{\Gamma^{\mu}}}]\!] & \stackrel{\text{def}}{=} & [\![P]\!]^*
\end{array}
$$

P^* loops on P, for any P. One-shot functions can be encoded in Γ^{μ} by merely omitting '*' operators.

2.3.3 Calculus of Gamma Programs (CGP)

CGP [37] was proposed as an alternative to Γ. CGP does not suffer from Γ's curious property, within an interleaving composition of programs, of being able to choose reduction of the inapplicable left-hand side of a sequential composition when an applicable primitive function is also available. CGP chooses to apply applicable primitive functions over all other kinds of programs. If no applicable primitive functions are available, then CGP will reduce sequential compositions, synchronously if more than one is available. If no applicable primitive functions and no sequential compositions are available (i.e. if the program is an interleaving composition of inapplicable primitive functions), then the whole program is synchronously rewritten to the empty program. Although CGP certainly does avoid the particular property its creators disliked in Γ, it also has its own curiosities. These shall be seen in Sections 2.5 and 2.4.

The SOS of CGP is given in [37] and below.

$$\frac{\exists \vec{m} \subseteq M.B\vec{m}}{\langle (B,A), M\rangle \Rightarrow \langle (B,A), M[A\vec{m}/\vec{m}]\rangle}\ \mathbf{App_1} \qquad \frac{\neg\exists \vec{m} \subseteq M.B\vec{m}}{\langle (B,A), M\rangle \Rightarrow M}\ \mathbf{App_2}$$

$$\frac{\langle P, M\rangle \Rightarrow \langle P', N\rangle}{\langle Q \circ P, M\rangle \Rightarrow \langle Q \circ P', N\rangle}\ \mathbf{Seq_1} \qquad \frac{\langle P, M\rangle \Rightarrow M}{\langle Q \circ P, M\rangle \Rightarrow \langle Q, M\rangle}\ \mathbf{Seq_2}$$

$$\frac{\langle P, M\rangle \Rightarrow M \quad \langle Q, M\rangle \Rightarrow M}{\langle P + Q, M\rangle \Rightarrow M}\ \mathbf{Int_1}$$

$$\frac{\langle P, M\rangle \Rightarrow \langle P, N\rangle}{\langle P + Q, M\rangle \Rightarrow \langle P + Q, N\rangle\ \&\ \langle Q + P, M\rangle \Rightarrow \langle Q + P, N\rangle}\ \mathbf{Int_2}$$

$$\frac{\langle P, M\rangle \Rightarrow \langle P', M\rangle \quad \langle Q, M\rangle \Rightarrow \langle Q', M\rangle \quad P' \neq P \quad Q' \neq Q}{\langle P + Q, M\rangle \Rightarrow \langle P' + Q', M\rangle}\ \mathbf{Int_3}$$

$$\frac{\langle P, M\rangle \Rightarrow M \quad \langle Q, M\rangle \Rightarrow \langle Q', M\rangle \quad Q' \neq Q}{\langle P + Q, M\rangle \Rightarrow \langle P + Q', M\rangle\ \&\ \langle Q + P, M\rangle \Rightarrow \langle Q' + P, M\rangle}\ \mathbf{Int_4}$$

CGP is translated into PT as follows:

$$i_{\mathrm{CGP}} \stackrel{\mathrm{def}}{=} (\mathtt{semi}, \mathtt{cond}, \mathtt{sync}, \mathtt{if})$$

where:
$(\mathbf{0} = \mathtt{if} = \mathtt{fail}) < \mathtt{semi} < \mathtt{sync} \leq (\mathtt{cond} = \mathtt{succ}) < \infty$

$$\begin{array}{lcl} [\![(B,A)_{\mathrm{CGP}}]\!] & = & (B, A, \mathcal{S}_\Gamma)^* \\ [\![P_2 \circ_{\mathrm{CGP}} P_1]\!] & = & [\![P_1]\!] \fatsemi [\![P_2]\!] \\ [\![P_1 +_{\mathrm{CGP}} P_2]\!] & = & [\![P_1]\!] \,|||\, [\![P_2]\!] \end{array}$$

Proposition 2 (Correctness of the translation) *Our translation of CGP into PT is correct.*

Proof by induction over the structure of the transition relation. The proof is to be found in Appendix A.2.

CGP$^\mu$: CGP with recursion and single-shot functions

The same weaknesses in the use of recursion that are present in Γ are present in CGP. We can easily define a new language, which we christen CGP$^\mu$, which exhibits the same context-sensitive and synchronisation behaviour as CGP, but with one-shot functions and general recursion.

$$\begin{array}{lcl} i_{\mathrm{CGP}^\mu} & \stackrel{\mathrm{def}}{=} & i_{\mathrm{CGP}} \\ [\![(B,A)_{\mathrm{CGP}^\mu}]\!] & \stackrel{\mathrm{def}}{=} & (B, A, \mathcal{S}_\Gamma) \\ [\![P_2 \circ_{\mathrm{CGP}^\mu} P_1]\!] & \stackrel{\mathrm{def}}{=} & [\![P_1]\!] \fatsemi [\![P_2]\!] \\ [\![P_1 +_{\mathrm{CGP}^\mu} P_2]\!] & \stackrel{\mathrm{def}}{=} & [\![P_1]\!] \,|||\, [\![P_2]\!] \\ [\![P^* \mathrm{CGP}^\mu]\!] & \stackrel{\mathrm{def}}{=} & [\![P]\!]^* \end{array}$$

CΓ: a CGP variant

We can define a variant of CGP in which successful function application is favoured over reduction of a sequential composition. If no functions can be successfully applied, then rewriting sequential compositions is performed non-synchronously and in preference to rewriting primitive functions. Primitive functions are rewritten synchronously, when none can be applied. CΓ is therefore identical to CGP except for its synchronisation behaviour.

$$i_{\mathrm{C}\Gamma} \stackrel{\text{def}}{=} \begin{array}{c}(\mathbf{0} = \texttt{if} = \texttt{fail}) < \texttt{sync} \leq \texttt{semi} \\ < (\texttt{succ} = \texttt{cond}) < \infty\end{array}$$

We present examples of programs executed according to the CΓ scheme in Section 2.4 and Section 2.5. As with Γ and CGP, we can easily define $\mathrm{C}\Gamma^{\mu}$.

2.3.4 A context-sensitive Gamma (CSΓ)

We define a new language called CSΓ, whose interleaving composition favours successful primitive function applications over all other kinds of reductions. However, unlike Γ and CGP, no synchronous rewriting of inapplicable programs occurs.

$$i_{\mathrm{CS}\Gamma} \stackrel{\text{def}}{=} \begin{array}{c}(\mathbf{0} = \texttt{if}) < \texttt{sync} \leq (\texttt{fail} = \texttt{semi}) \\ < (\texttt{succ} = \texttt{cond}) < \infty\end{array}$$

We present examples of programs executed according to the CSΓ scheme in Section 2.4 and Section 2.5. $\mathrm{CS}\Gamma^{\mu}$ is defined straightforwardly in an analogous manner to Γ^{μ}.

2.3.5 A context-insensitive Gamma (CIΓ)

We define a new language called CIΓ, whose interleaving composition is completely context insensitive. That is, it rewrites either of its arguments without examining the other (as long as neither of its arguments are ϵ: rewriting empty programs always takes priority over other rewrites).

$$i_{\mathrm{CI}\Gamma} \stackrel{\text{def}}{=} \begin{array}{c}(\mathbf{0} = \texttt{if}) \leq \texttt{sync} \leq \\ (\texttt{fail} = \texttt{semi} = \texttt{succ} = \texttt{cond}) < \infty\end{array}$$

We present examples of programs executed according to the CIΓ scheme in Section 2.4 and Section 2.5. As before, we can define a language $\mathrm{CI}\Gamma^{\mu}$, which offers recursion at arbitrary levels.

2.3.6 Alternative notions of resource consciousness 1: state sets

We can ensure that the initial multiset is transformed as a set. That is, multiple identical results are absorbed. If our initial multiset is a set, then $\mathcal{S}_{\cup}$ ensures that it remains so throughout the computation.

$$\mathcal{S}_{\cup}(B,A,\sigma) \;=\; \begin{cases} (\mathtt{succ},\mathtt{succ},\sigma \cup A\vec{\sigma}) & \text{if } \exists\vec{\sigma}\subseteq\sigma.B\vec{\sigma} \\ (\mathtt{fail},\mathtt{fail},\sigma) & \text{otherwise} \end{cases}$$

2.3.7 Alternative notions of resource consciousness 2: non-linearity

In [97], we observed that Γ function application is *linear* in the sense of Girard's linear logic [60]. That is, multiset elements are consumed when a function is applied to them. If multiple copies of an element are required, then that element must be explicitly copied. However, using an alternative selection function, we can remove this linearity constraint. Consider selection function $\mathcal{S}_{\uplus}$. If we think in terms of a computation as a chemical reaction, then $\mathcal{S}_{\Gamma}$ treats elements of the state as reagents, which are consumed in order to create their reaction products. In contrast, $\mathcal{S}_{\uplus}$ treats elements of the state as catalysts, being unchanged themselves and yet precipitating change.

$$\mathcal{S}_{\uplus}(B,A,\sigma) \;=\; \begin{cases} (\mathtt{succ},\mathtt{succ},\sigma \uplus A\vec{\sigma}) & \text{if } \exists\vec{\sigma}\subseteq\sigma.B\vec{\sigma} \\ (\mathtt{fail},\mathtt{fail},\sigma) & \text{otherwise} \end{cases}$$

2.3.8 AlChemy: chemical reactions modelled using λ-terms

An example model for whose encoding we can make use of the 'catalytic' reactions described above is AlChemy [51, 50, 52]. Alchemy was proposed as an abstract model for studying the evolution of self-organising and autocatalytic chemical solutions. An introduction to AlChemy is given in Section 1.2.7. We can easily build an AlChemy interpreter in PT, using the following encoding:

$$\begin{array}{lll} \mathcal{S}_{\text{AlChemy}}(B,A,\sigma) & = & (\mathtt{succ},\mathtt{succ},\sigma') \\ \text{where} & & \exists\vec{\sigma}\subseteq\sigma.B\vec{\sigma}\wedge \\ & & N=\sigma\uplus A\vec{\sigma}\wedge\sigma'\subseteq N\wedge |\sigma'|=|\sigma| \end{array}$$

$$\begin{array}{lll} i_{\text{AlChemy}} & = & \mathbf{0}=\mathtt{semi}=\mathtt{fail}=\mathtt{sync}=\mathtt{if}<\mathtt{succ} \\ [\![\text{alchemy interpreter}]\!] & = & (B,A,\mathcal{S})^* \\ \textit{where} & & \\ B(x,y) & = & \text{True} \\ A(x,y) & = & \{xy\} \\ \mathcal{S} & = & \mathcal{S}_{\text{AlChemy}} \end{array}$$

2.3.9 Abortion and deadlock

Primitive functions in Γ and CGP cannot abort or deadlock. However, in order to be as flexible as possible in the presence of aborting or deadlocking software components, we should ensure that our coordination language handles these situations gracefully. In this section, we give a few examples of $\mathcal{S}$ functions and i-tuples which give rise to different behaviours in the presence of aborting or deadlocking primitive functions.

Abortion

The following $\mathcal{S}$ function makes a primitive function abort if no sequence of elements in the set satisfies B:

$$\mathcal{S}_A(B, A, \sigma) = \begin{cases} (\texttt{succ}, \texttt{succ}, \sigma \cup A\vec{\sigma}) & \text{if } \exists \vec{\sigma} \subseteq \sigma . B\vec{\sigma} \\ (\texttt{fail}, \texttt{fail}, \bot) & \text{otherwise} \end{cases}$$

The behaviours of programs in interleaving compositions depend upon the i-tuple we have. If $\texttt{succ} > \texttt{fail}$ in the i-tuple, then non-aborting primitive functions are chosen over aborting primitive functions. In other words, non-abortion is favoured. This corresponds to *angelic* rewriting in the terminology of [11]. If $\texttt{succ} < \texttt{fail}$ in the i-tuple, then aborting primitive functions are chosen over non-aborting primitive functions. In other words, abortion is favoured. This corresponds to *demonic* rewriting in the terminology of [11]. Action Systems features abortion-favouring primitive functions.

Deadlocking

The following $\mathcal{S}$ function makes a primitive function deadlock if no sequence of elements in the set satisfies B:

$$\mathcal{S}_D(B, A, \sigma) = \begin{cases} (\texttt{succ}, \texttt{succ}, \sigma \cup A\vec{\sigma}) & \text{if } \exists \vec{\sigma} \subseteq \sigma . B\vec{\sigma} \\ (\texttt{fail}, \texttt{fail}, \Delta) & \text{otherwise} \end{cases}$$

Our choice of i-tuple determines the behaviours of interleaved programs, as before. If $\texttt{succ} > \texttt{fail}$, then non-deadlocking primitive functions are chosen over deadlocking primitive functions. Conversely, if $\texttt{succ} < \texttt{fail}$ in the i-tuple, then deadlocking primitive functions are chosen over non-deadlocking primitive functions.

Combining deadlock and abortion

Using variants of the above two selection functions, we can encode a language which features both aborting and deadlocking primitive functions:

$$\mathcal{S}_{A'}(B, A, \sigma) = \begin{cases} (\texttt{succ}, \texttt{succ}, \sigma \cup A\vec{\sigma}) & \text{if } \exists \vec{\sigma} \subseteq \sigma . B\vec{\sigma} \\ (\texttt{fail}_1, \texttt{fail}_1, \bot) & \text{otherwise} \end{cases}$$

$$\mathcal{S}_{D'}(B, A, \sigma) = \begin{cases} (\texttt{succ}, \texttt{succ}, \sigma \cup A\vec{\sigma}) & \text{if } \exists \vec{\sigma} \subseteq \sigma . B\vec{\sigma} \\ (\texttt{fail}_2, \texttt{fail}_2, \Delta) & \text{otherwise} \end{cases}$$

If $\texttt{succ} > \texttt{fail}_2 > \texttt{fail}_1$, then successful rewrites are chosen over all other rewrites and deadlocking rewrites are chosen over aborting rewrites.

2.4 Context sensitivity and interleaving choice

We compare four different versions of the interleaving operator: those of Γ, CGP, CSΓand CΠ. Figs. 2.3, 2.4 and 2.5 show the behaviours of the four composition operators when reducing two programs.

The figures display a stylised version of reduction and are meant to be read from top to bottom. Each tree indicates all of the possible reduction paths starting with some initial configuration. Initial and final configurations of a particular reduction are indicated by, respectively, the left-hand operand and right-hand operand of a '$\rightarrow$', appropriately oriented. Intermediate configurations containing ϵ have been elided in the interests of brevity. For each figure, we show the possible reduction paths generated using one-shot functions and the parameters of the languages being compared. We define the *behaviours* of a program P (written $\mathcal{B}(P)$) as a set of pairs. Each pair contains two multisets: the initial multiset and a final multiset which can be generated from the initial multiset by program P. If a program can generate several possible final multisets for a particular initial multiset, then the behaviours of that program will contain a pair for every possible final multiset. For example, the program P:

$$
\begin{array}{l}
(B, A, \mathcal{S}_\Gamma) \,|||\, (B', A', \mathcal{S}_\Gamma) \\
\text{where} \quad \begin{array}{lcl} Bx & = & \text{True} \\ Ax & = & \{1\} \\ B'x & = & \text{True} \\ A'x & = & \{2\} \end{array}
\end{array}
$$

when applied to the multiset $\{0\}$, has behaviours:

$$\mathcal{B}(P) \supseteq \{(\{0\},\{1\}),(\{0\},\{2\})\}$$

The behaviours of a program might well be different from those shown in the figures below, if primitive functions are used which are implicitly recursive (such as those in Γ). However, we wish here to highlight the program context-sensitive curiosities of the interleaving operators, which would still exist even if all primitive functions were implicitly recursive. In the examples given, therefore, we use non-recursive primitive functions to keep the diagrams manageable. As in Chapter 1, we use Pnm to indicate a primitive function ($\lambda x.x = n, \lambda x.\{m\}$) (i.e. a primitive function that converts an n into an m, if an n is found in the multiset).

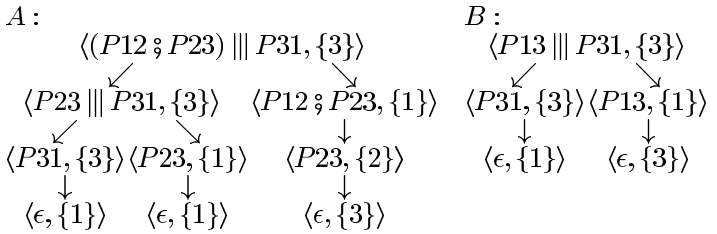


Figure 2.3: Programs A and B, reduced in PT with CII's parameters. Note that A and B yield the same behaviours.

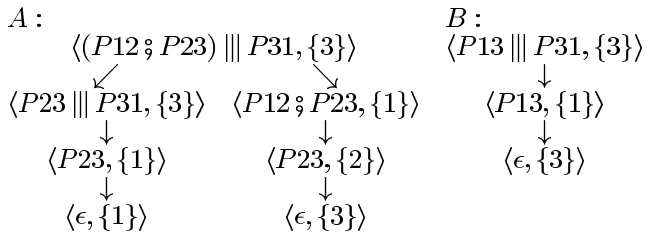


Figure 2.4: Programs A and B, reduced in PT with Γ's parameters. Note that A and B yield different behaviours, even though twos produced by $P12$ cannot be consumed by $P31$.

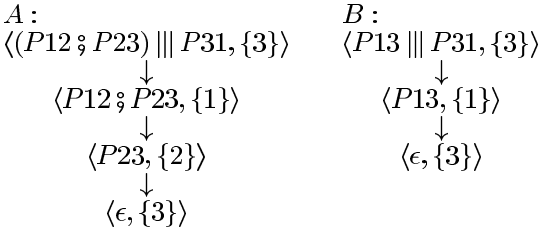


Figure 2.5: Programs A and B, reduced in PT with CSΓ or CGP's parameters. Note that A and B yield the same behaviours, as in the CIΓ scheme.

The figures illustrate that CIΓ, CSΓ and CGP are well-behaved for programs A and B in the sense that they produce the same behaviours for both programs A and B. This is certainly what we would expect from an interleaving composition operator, as the intermediate results produced and consumed by $P12;P23$ (namely twos) *can be neither generated nor consumed by* $P31$. Therefore, we should be surprised if the fact that A takes twice as many reductions as B to turn ones into threes influences the possible outcomes of the computation. However, this is *exactly* what happens for the Γ version, which seems to suggest that Γ is badly-behaved in some sense. The reason why, in Γ, the behaviours of program A differ from those of B is that, in Γ, a reduction inside a sequential composition is just as interesting as a successful application of a primitive function—as witnessed by the fact that `semi` = `succ` in Γ. Therefore, in an interleaving composition, we can always choose to reduce inapplicable primitive functions in a sequential composition, regardless of whether another component of the interleaving composition is applicable. However, we *cannot* reduce inapplicable primitive functions in an interleaving composition, unless no other reductions are possible. We believe that this asymmetry between primitive functions and sequentially composed functions should be considered curious at best.

2.5 Synchronous and non-synchronous reductions

A synchronous reduction may occur when $\texttt{fail} \leq \texttt{sync}$. In such a case, one or both operands of the interleaving operator are rewritten, without altering the state. If $\texttt{succ} \leq \texttt{sync}$, then all interleaving compositions rewrite to either a single primitive function or to the empty program without any transformations being performed on the multiset. We regard this behaviour as degenerate and do not examine it further.

We attempt to give the reader intuitions on the impact on a formalism of choosing different synchronisation strategies, by taking the parameters of a particular formalism and investigate the effects of altering `sync`. In Figs. 2.6, 2.7, 2.8 and 2.9, we investigate the impact on the behaviours of programs, using the parameters (except for `sync`) of, respectively, CIΓ, Γ, CSΓ and CGP.

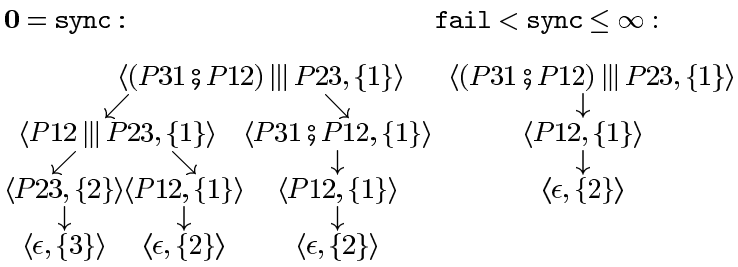


Figure 2.6: The reduction paths possible using CIΓ's parameters. The two approaches to synchronisation yield different behaviours, for this program.

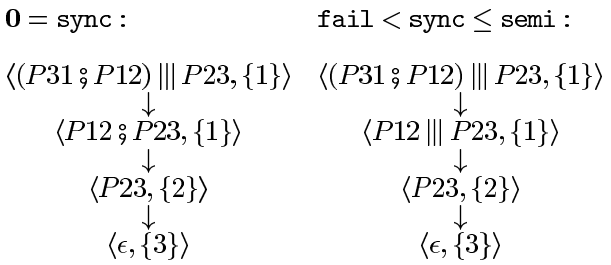


Figure 2.7: The reduction paths possible using Γ's parameters. The two approaches to synchronisation yield the same behaviours, for this program. The result is generalised to all programs in Section 2.5.1.

Notice that making Γ FS ($\texttt{fail} \leq \texttt{sync}$) does not affect the behaviours of the program, unlike for CSΓ and CIΓ. We can justify this informally as follows:

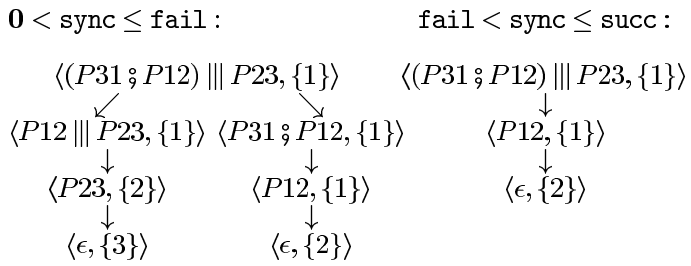


Figure 2.8: The reduction paths possible using CSΓ's parameters. The two approaches to synchronisation yield different behaviours, for this program.

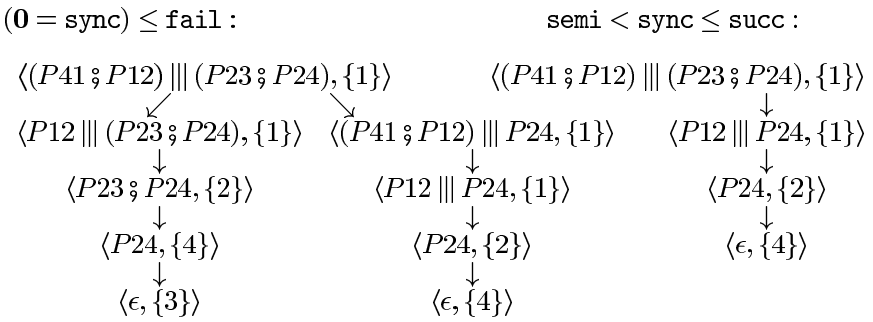


Figure 2.9: The reduction paths possible using CGP's parameters. The two approaches to synchronisation yield different behaviours, for this program.

Because $\texttt{fail} < (\texttt{semi} = \texttt{succ})$ in Γ, then the only occasion on which all the programs in an interleaving composition yield `fail` is when all of them are simple (i.e. do not include sequential composition [65]) and none of them can be successfully applied to the multiset. In such a case, it makes no difference whether the primitive functions are reduced synchronously, because applying them in any order or all at once will not change the multiset or make one of the other primitive functions applicable.

The argument can be made formal by a straightforward induction.

In contrast, CSΓ and CIΓ *can* behave differently when augmented with synchronisation on failure. CGP favours (synchronised) reductions of sequentially composed programs over (synchronised) reductions of primitive functions. CSΓ does not favour reductions of sequentially-composed programs over reductions of primitive functions.

2.5.1 Some results linking Γ, Γ^μ, CGP and CGP$^\mu$

We conclude this section by giving a number of results linking the translations of Γ, Γ^μ, CGP and CGP$^\mu$.

Proposition 3 *All* Γ *programs can be translated into* Γ^{μ}.

Proof: Consider the following translation of Γ into Γ^{μ}:

$$\begin{array}{lcl} [\![(B,A)_{\Gamma}]\!] & = & (B,A)^{*}_{\Gamma^{\mu}} \\ [\![Q \circ_{\Gamma} P]\!] & = & [\![Q]\!] \circ_{\Gamma^{\mu}} [\![P]\!] \\ [\![P +_{\Gamma} Q]\!] & = & [\![P]\!] +_{\Gamma^{\mu}} [\![Q]\!] \end{array}$$

This translation is sound. Proof by induction over the structure of the programs. The proof relies upon the fact that $\infty > (\texttt{succ} \sqcup \texttt{fail} \sqcup \texttt{semi})$, After a successful or unsuccessful application of the translation of a primitive function (B,A), that translation is always of the form $\epsilon \overset{\mathbf{c}}{\triangleright} (B,A)^{*}$. Therefore, PT always makes the transition $\langle P,M\rangle \overset{(\mathbf{0},\infty)}{\Longrightarrow}_{PT} \langle P',M\rangle$, which either regenerates the primitive function (if the application was successful) or discards it (if the application was unsuccessful). □

Proposition 4 *All CGP programs can be written in CGP*$^{\mu}$.

Proof: Consider the following translation of CGP into CGP$^{\mu}$:

$$\begin{array}{lcl} [\![(B,A)_{\mathrm{CGP}}]\!] & = & (B,A)^{*}_{\mathrm{CGP}^{\mu}} \\ [\![Q \circ_{\mathrm{CGP}} P]\!] & = & [\![Q]\!] \circ_{\mathrm{CGP}^{\mu}} [\![P]\!] \\ [\![P +_{\mathrm{CGP}} Q]\!] & = & [\![P]\!] +_{\mathrm{CGP}^{\mu}} [\![Q]\!] \end{array}$$

This translation is sound. Proof by induction over the structure of the programs. □

Proposition 5 *Simple [65] programs have the same set of behaviours in* Γ *and CGP.*

Proof: Simple inductive proof on the structure of the programs. The proof hinges on the fact that the only difference between i_{Γ} and i_{CGP} is in the relative values of `semi`. Therefore, we can easily show that programs which do not include sequential composition must generate the same sets of behaviours. □

Proposition 6 *There is a program which is both a* Γ^{μ} *program and a CGP*$^{\mu}$ *program, but neither a* Γ *nor a CGP program.*

Proof: Consider the following program schema. For all A,

$$(\mathrm{True}, A)$$

Such programs can be applied once to the empty multiset and then terminate, yielding a multiset A. Such programs cannot be written in Γ [127] or in CGP. They contain no sequential composition or interleaving operators. The parameters for the interleaving and sequential composition operators are the only places in which Γ^{μ} and CGP$^{\mu}$ differ. Therefore, the behaviours of these programs will be identical in Γ^{μ} and in CGP$^{\mu}$. □

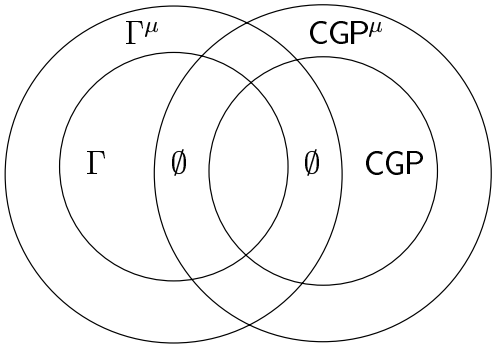


Figure 2.10: A comparison between Γ, CGP, Γ^μ and CGP^μ. Set intersection indicates the existence of programs which are both syntactically identical and have the same behaviours when reduced in each formalism.

Proposition 7 *There is no program which is both a Γ^μ and a CGP program but not a Γ program.*

Proof (by contradiction): Assume that there is such a program P. As P is a Γ^μ program but not a Γ program, it must either (i) contain no recursion or (ii) contain sequential composition within the scope of recursion. If (i), then P is not a CGP program, because CGP (like Γ) has recursive primitive functions. If (ii), then it cannot be a CGP program because sequential composition in CGP behaves differently to sequential composition in Γ. So either way, the program is not a CGP program. □

Proposition 8 *There is no program which is both a CGP^μ and a Γ program but not a CGP program.*

Proof: Almost identical to that of proposition 7. □

The results established above enable us to present the diagram in Fig. 2.10, which relates Γ, Γ^μ, CGP and CGP^μ.

2.6 A taxonomy of Γ-like languages

We have described a number of multiset transformation languages in terms of the levels of interest which they associate with different reductions. A graphical representation of the i-tuples associated with each language is shown in Fig. 2.11. The explanation for the double appearance of Γ is given in Section 2.5.

2.7 Fairness

The languages for which translations have been given here make no fairness assumptions. Adding to PT either a fairness assumption or a random assignment statement would render possible sound translations of languages

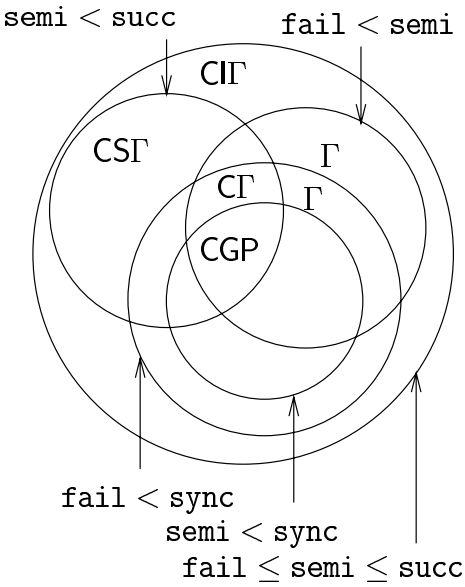


Figure 2.11: A taxonomy of multiset transformation languages. The diagram is formed by classifying according to the relationships between members of the i-tuple of each language discussed in this chapter. Set intersection indicates the existence of programs which are both syntactically identical and have the same behaviours when reduced in each formalism. We omit references to `cond` in our classification as all our examples make `cond` = `succ`. All of the languages discussed use selection function $\mathcal{S}_\Gamma$. The 'μ' versions of the languages are to be found at the same places in the diagram as their 'non-μ' counterparts. The explanation for Γ's double appearance is given in Section 2.5. The empty set intersections are explained in the text.

requiring fairness [4]. Examples of such languages include UNITY and Action Systems, both of which were described in Chapter 1. We leave such investigations to future work.

2.8 Future work

The work here presented constitutes the first steps in an analysis of languages for parallel programming. We should extend our range of examples by attempting to translate more formalisms into PT. We should also investigate the possibility of including a truly parallel composition operator in PT. Our interleaving operator can perform synchronous reductions, but only by throwing away the results of the computations performed by both its operands. We plan to investigate ways of encoding state splitting and recombination which could allow non-degenerate reductions to occur in lockstep. Furthermore, adding to PT either a fairness assumption or a random assignment statement would render possible sound translations of languages

requiring fairness [4]. We should also investigate the properties of PT independently of the parameters which might be chosen, thereby attempting a work in the spirit of that undertaken in [65] for Γ.

Currently, PT contains two forms of conditional. One hides in the selection function and evaluates the predicate B from a primitive function. The other lives in programs in the form $(P \overset{\mathbf{n}}{\triangleright} Q : R)$. This is inelegant. The ideal solution would be to find a way of allowing only a single form of conditional. However, our efforts in this direction have so far proved unfruitful.

Finally, no multiset transformation languages are known or suggested which fit into the blanks in Fig 2.11. We will have to wait for future work to tell us whether the denizens of these places are languages worthy of study in their own right.

2.9 Conclusions

We have offered PT, a generalised and parameterisable operational semantics for parallel programming. Instantiating the semantics in different ways results in different languages.

We have used PT to discuss the (dis-) similarities between a number of multiset transformation languages which differ in the ways in which they handle interleaving compositions of functions. Some of the languages compared existed before PT (Γ and CGP), while others have been proposed as a result of examinations of the parameters used in the encodings of Γ and CGP(e.g. CΓ, CIΓ and CSΓ). Our encodings delivered insights into the context-sensitivity of different interleaving compositions and suggested weaknesses in Γ's and in CGP's semantics.

We believe that the amenability of our semantics as a target language for translation demonstrates the utility of quantifying the level of interest associated with different reductions. Another example of the utility of our approach can be seen when we consider that Ciancarini *et. al.* claimed that Γ lacked the context-sensitive reduction property because of the way in which the interleaving composition was defined [37]. We have shown that making `semi` = `fail` effectively redefines sequential composition by removing the '⊔' function on flags and makes a CS variant of Γ. Therefore, a CS version of Γ can be defined *either* (such as in Ciancarini's treatment) by redefining the interleaving composition *or* (in our treatment) by defining a new sequential composition operator for Γ, whose reductions are less interesting to interleaving composition.

Finally, both Γ and CGP mix properties of context-sensitive reduction and synchronisation on failure. We believe that our analysis of both of these properties in terms of PT shows that there is some mileage to be gained by explicitly separating the two issues.

Chapter 3

State synchronisation and data contexts: the road to L is paved with good extensions[1]

.

In this chapter, we capture synchronisation and data contexts in PT, the model for parallel reduction introduced in the previous chapter. Data contexts allow us access to tuples' values without having to remove them from the multiset. Synchronisation control allows us to specify when changes to the multiset are 'fixed' and therefore made visible to the program. Alone or in combination, these properties allow us to write multiset transformation (MST) programs which behave in a greater variety of ways than those previously possible. Although excluded from our earlier analyses, these two extensions are needed for a flexible and applicable parallel reduction system: many physical and biological systems exist whose reduction possesses these properties. Examples include Lindenmayer systems (L-systems) and cellular automata (CAs). We show how to capture, in PT, state synchronisation, data context properties and their combination, giving a number of motivating examples.

3.1 Introduction

In Chapter 2, we described PT, a unified theoretical framework for 'parallel' state transformations. The main feature of the model is the presence of parameters, the substitution for which enforces certain constraints on the dynamics of reduction. Substituting different parameters allows (features of) different languages to be encoded. PT is therefore a tool for comparing

[1]This chapter represents joint work with Professor Przemyslaw Prusinkiewicz of the University of Calgary, Alberta, Canada. I would like gratefully to acknowledge the NWO, whose grant SIR 12-2643 partially supported me during my visit to Canada.

a number of extant formalisms within a single framework. This enables us to study the effects of different control-flow operators either alone or in combination, without having to compare complete alternative languages.

Despite the triumphs of Chapter 2 in clarifying program context-sensitive interleaving operators, no explicitly parallel operators were examined. While interleaving is often considered an adequate approach to theoretical treatments of parallelism, we were motivated to attempt to encapsulate, in PT, true parallel operators, whether prescriptive (you *must* reduce in parallel if you can) or non-prescriptive (you *may* reduce in parallel if you can). In particular, we wished to encode, in a straightforward way, the prescriptive, synchronised, (data) context-sensitive parallel reduction of systems such as Lindenmayer systems [91] and Cellular Automata [145]. In such systems, all elements of the state (usually array elements) are updated logically synchronously, using the values of (some of) their nearest neighbours to calculate their next state. Such systems are described in Chapter 1. As we demonstrate in Chapter 5, these systems can be encoded in MSTs with interleaving operators, but the encoding is awkward. Specifically, it is difficult to synchronise the reductions so that all members of one generation are updated before any members of the next generation. This problem is particularly acute when elements need to observe the values of other elements, which are not to be removed from the multiset. Such elements we refer to as *data context*. For example, if reductions are logically in parallel, elements can be both removed and examined as data context logically simultaneously (See Figure 3.1). In this chapter, we explore encodings of data context-sensitivity and of state synchronisation in isolation and, finally, encode the combination of these two properties. To show that our new formalism (viz. a data context-sensitive reduction system with explicit state synchronisations) is a strict extension of those formalisms studied in Chapter 2, we prove that Γ [65] can be correctly translated into the combined formalism, with the appropriate choice of i-tuple. Throughout, our exposition is aided by examples.

This chapter is structured as follows. In Section 3.2, we introduce data contexts to PT. In Section 3.3, we introduce state synchronisation to PT, followed in Section 3.4 by examples of prescriptive and non-prescriptive parallel operators, captured using our mechanism. In Section 3.5, we combine data contexts and synchronisation and show the advantages of such a combination. In Section 3.6, we give a translation of Γ into the combined formalism. On the basis of our experiences in the last two chapters, we introduce the language Goblin, a multiset transformation language (MSTL) containing explicit state synchronisations and data contexts and suffering from none of the compositionality problems of Γ [17] or CGP [37], which were described in the previous chapter. Finally, we indicate possible avenues for future research and draw our conclusions.

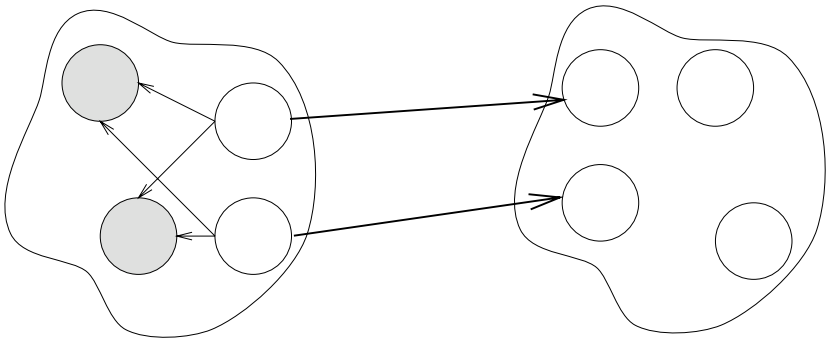


Figure 3.1: Two reductions, with overlapping data contexts which can be performed in parallel. The contextual arguments are shown as light grey; the non-contextual arguments as white. Arrows indicate which contextual elements are associated with each non-contextual element. Rewrites are written as arrows from multiset to multiset.

3.2 Data contexts in PT

Most MSTLs (e.g. Γ [20], CGP [37], CΓ [96]) remove a primitive function's arguments from the multiset when that function is successfully applied to the state (i.e. its predicate evaluates to true). We frequently find that our programs are somewhat lengthened by this restriction: in particular, we are sometimes forced to remove elements from the multiset then replace them, unchanged, just so that we can see their values [98]. This is awkward and is prone to encourage programmer errors. Contrast this state of affairs with that of Linda [58], which allows one to read the value of a datum in the tuple space, without removing it. The values of these contextual elements can be used to calculate the results of a function application. Our earlier work on PT [96] did not include any encoding of data contexts, but they can be encoded straightforwardly, by choosing the appropriate selection functions (selection functions are explained in Chapter 2). We give an example of such a selection function:

$$\mathcal{S}_C(B, A, \sigma) \quad \stackrel{\text{def}}{=} \quad \begin{cases} (\texttt{succ}_1, \texttt{succ}_2, \sigma[A(\vec{\sigma_1}, \vec{\sigma_2})/\vec{\sigma_1}]) \\ \qquad\qquad \text{if } \exists \vec{\sigma_1} \subseteq \sigma . \exists \vec{\sigma_2} \subseteq (\sigma - \vec{\sigma_1}) . B(\vec{\sigma_1}, \vec{\sigma_2}) \\ (\texttt{fail}_1, \texttt{fail}_2, \sigma) \qquad\qquad \text{otherwise} \end{cases}$$

Selection function S_C states that, when a primitive function is applied, we remove two disjoint subsets from the multiset. Condition B checks the values of both sequenced subsets, but action A only replaces the non-contextual elements of the multiset. Therefore, the contextual elements remain, undeleted and unchanged, in the multiset.

With the availability of data context comes the possibility of writing programs which perform the same function in different ways, depending

upon whether they make use of data context or not. Consider these two versions of a maximum function on multisets (the example comes originally from [16]). In what follows, we write:

$$(B(\vec{x},\vec{y}), A(\vec{x},\vec{y}))$$

For a (nameless) primitive function whose B and A take arguments $\vec{x}$ and data context $\vec{y}$. We write () for the empty argument list or empty context.

$$\begin{array}{lcll} \max_n & \stackrel{\text{def}}{=} & (B, A) & \\ & & \text{where} & \\ & & B((x,y),()) & = & x \leq y \\ & & A((x,y),()) & = & \{x\} \end{array}$$

$$\begin{array}{lcll} \max_c & \stackrel{\text{def}}{=} & (B, A) & \\ & & \text{where} & \\ & & B((x),(y)) & = & x \leq y \\ & & A((x),(y)) & = & \emptyset \end{array}$$

The behaviour of programs depends upon the relative values of the labels in the selection function and of those labels given as parameters to PT's SOS. The way in which this control is exerted is explained in Chapter 2. Assuming that $(\texttt{fail}_1 = \texttt{fail}_2 = \texttt{sync} = \texttt{if}) < (\texttt{succ}_1 = \texttt{succ}_2 = \texttt{cond})$, then it is easy to demonstrate that $\langle \max_n^*, M \rangle$ and $\langle \max_c^*, M \rangle$ yield the same results, for all multisets M.

In the above example, the number of arguments which must be *removed* from the multiset is one in the case of $\max_c$ and two in the case of $\max_n$. Therefore, for $\max_c$, a copy of the whole multiset can be given as data context to each process, together with that element which the process is supposed to rewrite. As there are n elements which are to be rewritten, and as only one (plus the data context) is required at each process, n processes can reduce in parallel. For $\max_n$, two elements (and no data context) are required at each process before a comparison can be performed. Therefore, we should expect the maximum possible parallelism for $\max_n$ (respectively $\max_c$) to be $n/2$ (respectively n) for a multiset of cardinality n. If this is indeed the case in general, then we have another strong motivation for using data contexts, where this is possible. Unfortunately, in the absence of proper experimental evaluation, we cannot yet claim that data contexts can make implementations of a multiset transformation language more efficient. We leave this avenue for future research.

3.3 Synchronous state update in PT

Now that we have shown how to encode data contexts in PT, we turn to the second of our tasks, to encode different state update policies. State update can occur after every primitive function application, or after a number of primitive function applications. If it occurs after each primitive function

application, then an interleaving operator will perform state updates in an interleaving fashion. If state synchronisation occurs *after* a number of primitive function applications, then an interleaving operator can behave as a truly parallel operator. Examples will be given in due course. At first, we encode different state update policies without the presence of data contexts. Later, in Section 3.5, we combine state synchronisation and data context sensitivity.

The semantics of Γ's selection function implies that a successful function application alters the state immediately. Furthermore, Γ offers an interleaving program composition [65] but no truly parallel program composition. Therefore, if logically parallel reductions are desired, an implementation will have to determine when it is safe to do these. As PT was originally designed as a extension of Γ, PT also makes it impossible explicitly to express logically simultaneous state updates.

Although interleaving composition is often used as a model of parallelism [110, page 48] and [114, page 36], there are situations wherein it is not a good model. We give examples in Section 3.5. If we wish to be able easily to encode these examples, we must capture the behaviour of a true parallel operator in PT. Both components of a true parallel operator can update the state logically simultaneously. Some parallel operators are also *prescriptive*, in that they demand that reductions which can be performed in parallel *must* be performed in parallel. Non-prescriptive parallel operators make it possible that reductions which *can* be performed in parallel are not so performed.

We claim that the essential difference between a parallel operator and an interleaving operator is in the synchronisation of state updates: when state updates happen after every function application, then reductions are logically interleaved: if state update occurs after a number of function applications, then reductions are logically in parallel. We should expect, then, that we can capture a parallel operator in PT by separating the state update ('fixing' the latest changes to the state) from primitive function application. This is, in fact, what we do, using the selection functions below:

$$\mathcal{S}_{S1}(B,A,(\sigma,\rho)) \stackrel{\text{def}}{=} \begin{cases} (\mathtt{succ}_1,\mathtt{succ}_2,(\sigma - \vec{\sigma_1}, \rho \uplus A(\vec{\sigma_1}))) & \text{if } \exists \vec{\sigma_1} \subseteq \sigma . B(\vec{\sigma_1}) \\ (\mathtt{fail},\mathtt{fail},(\sigma,\rho)) & \text{otherwise} \end{cases}$$

$$\mathcal{S}_{S2}(B,A,(\sigma,\rho)) \stackrel{\text{def}}{=} (\mathtt{fail},\mathtt{succ}_3,(\sigma \uplus \rho,\emptyset))$$

In each function, states are represented as pairs of multisets, the left-hand of which represents the available arguments and the right-hand of which represents the results of previous function applications. When a program terminates, we regard the left-hand multiset as the result of the computation. $\mathcal{S}_{S1}$ states that primitive functions search for elements with the required properties in the left-hand multiset (the argument multiset). If such elements are found, then they are removed from the left-hand set and the action function A is applied to them. The resulting multiset is combined using a multiset union with the right-hand side multiset (the result multiset). Selection function $\mathcal{S}_{S2}$ merely adds all of the elements of the right-hand

multiset to the left-hand multiset and empties the right-hand multiset. The effect of this is a state synchronisation: all of the changes made to the right-hand multiset are 'fixed' in the left-hand multiset and can therefore be used as function arguments for future primitive functions.

Using S_{S2}, defined above, we can define a synchronisation function s as follows, for any B, A:

$$\mathrm{s} \stackrel{\mathrm{def}}{=} (B, A, \mathcal{S}_{S2})$$

The synchronisation function is a primitive function and can therefore be used in a program just like any other primitive function. The form of this function is somewhat confusing, for historical reasons[2]. PT was introduced with primitive functions having the syntax $(B, A, \mathcal{S})$, for some $B, A, \mathcal{S}$ (see Chapter 2). In other words, all primitive functions possess a predicate (B) and an action function (A), as well as a selection function ($\mathcal{S}$). However, our synchronisation function s requires neither B nor A, which can therefore be instantiated in any way we please without altering s's behaviours.

Encoding state synchronisation using an explicit primitive function gives to the programmer the ability to state when, precisely, state updates take place. This, in turn, enables her to write programs which exhibit a wide variety of behaviours. For example, by placing a synchronisation function after every primitive function application, the programmer can force a Γ-like behaviour in which every function result is 'immediately' placed back into the (argument) multiset. On the other hand, she can ensure a logically maximally parallel execution by postponing synchronisation until after all possible reductions have been performed on the original multiset. Furthermore, we can capture a hybrid, a non-prescriptive parallel operator, similar to that in Schedules [36], wherein multiple reductions *may* be performed before a state update. We illustrate each of these claims, with examples, below.

Examples of synchronous and non-synchronous reductions

To give the reader a concrete example of the effect of synchronisation on multiset rewriting, consider the following examples. Both examples make use of the function (+1), which is defined below. The i-tuple which is used by both examples is also given below (i-tuples are explained in Chapter 2):

$$\begin{array}{lcl} (+1) & \stackrel{\mathrm{def}}{=} & (B, A, \mathcal{S}_{S1}) \\ \text{where} & & \\ B((x), ()) & = & \text{True} \\ A((x), ()) & = & \{x + 1\} \end{array}$$

$$\begin{array}{rcl} i & \stackrel{\mathrm{def}}{=} & (\mathtt{fail}_1 = \mathtt{fail}_2 = \mathtt{sync}) \\ & < & (\mathtt{succ}_1 = \mathtt{succ}_2 = \mathtt{semi}) \\ & < & \mathtt{succ}_3 \end{array}$$

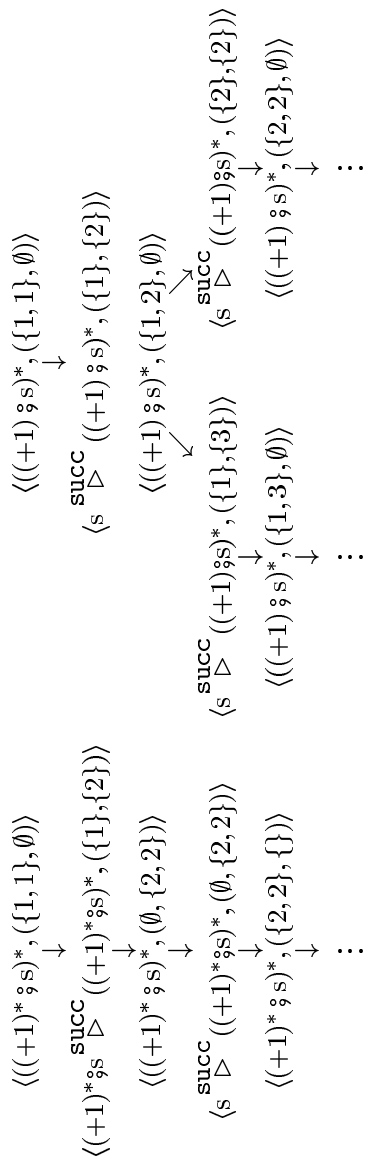


Figure 3.2: The possible reductions of (+1), under different state synchronisation strategies. The left-hand example synchronises the state only when all possible function applications have been performed. The right-hand example synchronises the state after every primitive function application.

The possible reductions, for a particular initial multiset, are shown in Figure 3.2. On the left-hand side, synchronisation occurs only when the function has been applied to the multiset as many times as possible. On the right-hand side, the state is synchronised after every primitive function application. The difference between the two examples is clear: the left-hand side's state synchronisations (updates) occur in lockstep, resulting in synchronous increases of all the multiset's elements. On the right-hand side, the behaviour is much less regular, with some multiset elements being updated repeatedly while others are untouched.

3.4 Encoding parallel operators

PT contains no true parallel operator: the closest that we have is an interleaving operator. However, using the notion of state synchronisation, we can encode different kinds of logically parallel operator into PT. The differences between the operators result from varying either the relative values of the elements of the i-tuple, or by offering different encodings of primitive functions (or both). In this section, we offer several different parallel operators, with examples.

3.4.1 A non-prescriptive parallel operator

When more than one reduction is possible, a non-prescriptive parallel operator may behave like an interleaving operator or like a true parallel operator. A good example of a non-prescriptive parallel operator can be found in Chaudron's Schedules language [36]. To make our interleaving operator behave in this fashion, we can use $\mathcal{S}_{S1}$ and $\mathcal{S}_{S2}$ and make $\texttt{succ}_3 = \texttt{succ}_2$ in i. In other words, a synchronisation under $\mathcal{S}_{S2}$ is no more interesting than a successful application of a primitive function. Therefore, when both a synchronisation and a (successful) primitive function application are offered in an interleaving composition, either can be chosen.

In what follows, we write $P12$ for a primitive function which turns a 1 into a 2 and $P\{1,2\}4$ for a primitive function which turns either a 1 or a 2 into a 4. Figure 3.3, gives an example of a non-prescriptive parallel operator. Some intermediate reduction steps are omitted for reasons of space.

3.4.2 A prescriptive parallel operator

A prescriptive parallel operator must reduce programs in parallel, if it can. The following definition of a Γ-like language ensures that all the functions offered in parallel are applied as many times as possible to the state before a synchronisation is allowed to occur. We use $\mathcal{S}_{S1}$ and $\mathcal{S}_{S2}$ as our selection functions.

[2] Thus is born legacy theory.

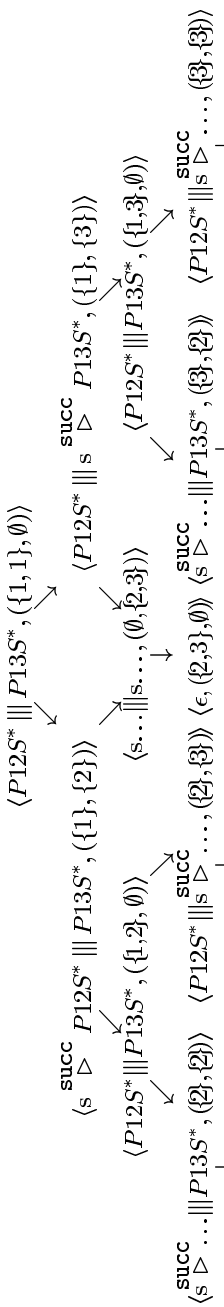


Figure 3.3: An example of a non-prescriptive, logically parallel composition of two functions. The state can be updated (synchronised) after *every* primitive function application or after a number of such applications.

$$\begin{array}{lll} [\![(B,A)]\!] & \stackrel{\text{def}}{=} & ((B,A,\mathcal{S}_{S1})^* \fatsemi \mathrm{s})^* \\ [\![P \fatsemi Q]\!] & \stackrel{\text{def}}{=} & [\![P]\!] \fatsemi [\![Q]\!] \\ [\![P \parallel Q]\!] & \stackrel{\text{def}}{=} & [\![P]\!] \mid\mid\mid [\![Q]\!] \end{array}$$

In Figure 3.4, we give the possible reductions, for a particular initial set, when $(\texttt{fail} = \texttt{sync}) < (\texttt{cond} = \texttt{succ}_1 = \texttt{succ}_2 = \texttt{succ}_3 = \texttt{semi})$. We write $PxyS^*$ for $(Pxy^* \fatsemi \mathrm{s})^*$. Some reduction steps have been omitted.

Prescriptive parallel operators are unable to exhibit some of the behaviours of non-prescriptive operators. Consider the following example. Again, some steps are omitted.

$$\begin{array}{c} \langle ((P12^* \fatsemi \mathrm{s})^* \mid\mid\mid (P\{1,2\}4)^* \fatsemi \mathrm{s})^*, (\{1,1\},\emptyset)\rangle \\ \downarrow \\ \langle ((P12^* \fatsemi \mathrm{s})^* \mid\mid\mid (P\{1,2\}4)^* \fatsemi \mathrm{s})^*, (\{1\},\{2\})\rangle \\ \downarrow \\ \langle ((P12^* \fatsemi \mathrm{s})^* \mid\mid\mid (P\{1,2\}4)^* \fatsemi \mathrm{s})^*, (\emptyset,\{2,2\})\rangle \\ \downarrow \\ \langle \mathrm{s} \overset{\mathbf{o}}{\triangleright} ((P12^* \fatsemi \mathrm{s})^* \mid\mid\mid (P\{1,2\}4)^* \fatsemi \mathrm{s})^*, (\emptyset,\{2,2\})\rangle \\ \downarrow \\ \langle ((P12^* \fatsemi \mathrm{s})^* \mid\mid\mid (P\{1,2\}4)^* \fatsemi \mathrm{s})^*, (\{2,2\},\emptyset)\rangle \\ \downarrow \\ \langle \epsilon, (\{2,2\},\emptyset)\rangle \end{array}$$

It is never possible for $P\{1,2\}4$ to reduce, because synchronisation can only occur when neither composed program can do anything more. $P12$ can continue to reduce until all the ones are consumed, by which point $P\{12\}4$ can do nothing.

3.5 Combining state synchronisation and data contexts

In previous sections, we introduced encodings which capture data contexts and state synchronisation in PT. In this section, we examine their combination, in two new selection functions. We notice that adding data context to multiset transformation in combination with synchronisation makes certain results, possible under a truly parallel composition, impossible under an interleaving composition. Therefore, in the presence of data context and state synchronisation, interleaving composition of functions is not the same as parallel composition of functions. So while interleaving is fine as a model of (non-prescriptive) parallelism when no data context is present, it fails as a model of parallelism when data context is present: it synchronises the state too often to maintain correctly the data context.

We show the advantage of our approach over a truly parallel operator, where both composed programs reduce simultaneously. In particular, we

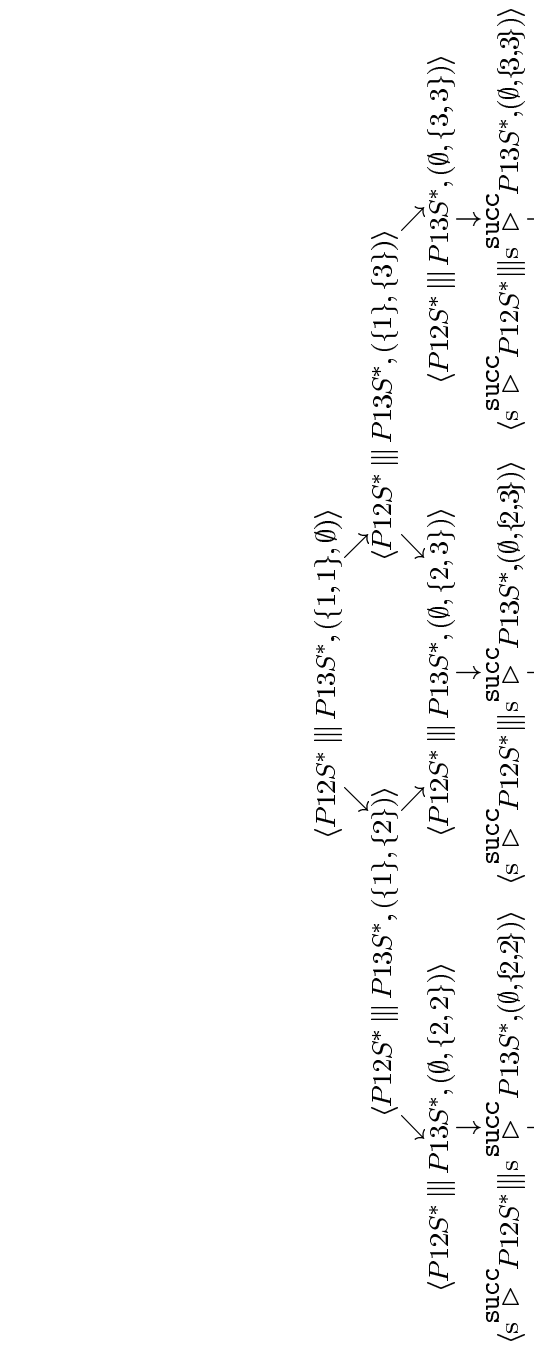


Figure 3.4: An example of a prescriptive, logically parallel composition of two functions with each other and with themselves. The state is updated (synchronised) only after all possible primitive functions have been applied.

show that encoding parallelism as interleaving plus explicit state synchronisation plus data contexts does not enforce a strict separation of the states of two programs which are reduced logically simultaneously. Instead, the data context required by one program may overlap with that required by the other program. In practice, implementations of L-systems [91, 115], cellular automata [145] and many other scientific applications require precisely this kind of overlap between distributed sub-states to realise their logically parallel reductions. This in itself makes a formal model of logically parallel reduction with overlapping states a desirable goal.

One of the advantages of our approach is that it does not require that a program be explicitly written as reducing in parallel with itself in order to be so reduced. If state synchronisation happens *after* a recursive program has been performed n times, the result is the same as if the program had been reduced n times in parallel with itself.

Combining state synchronisation and data context-sensitivity is not entirely straightforward. First of all, consider a naïve combination, thus:

$$\mathcal{S}_{SC1}(B, A, (\sigma, \rho)) \stackrel{\text{def}}{=} \begin{cases} (\mathtt{succ}_1, \mathtt{succ}_2, (\sigma - \vec{\sigma_1}, \rho \uplus A(\vec{\sigma_1}, \vec{\rho_1}))) \\ \qquad\qquad \text{if } \exists \vec{\sigma_1} \subseteq \sigma, \exists \vec{\rho_1} \subseteq (\rho - \vec{\sigma_1}).B(\vec{\sigma_1}, \vec{\rho_1}) \\ (\mathtt{fail}_1, \mathtt{fail}_2, (\sigma, \rho)) \qquad\qquad \text{otherwise} \end{cases}$$

$$\mathcal{S}_{SC2}(B, A, (\sigma, \rho)) \stackrel{\text{def}}{=} (\mathtt{fail}_1, \mathtt{succ}_2 + 1, (\sigma \uplus \rho, \emptyset))$$

We define a new synchronisation function s′ as follows, making use of S_{SC2}, defined above. For any B, A:

$$\mathrm{s}' \stackrel{\text{def}}{=} (B, A, \mathcal{S}_{SC2})$$

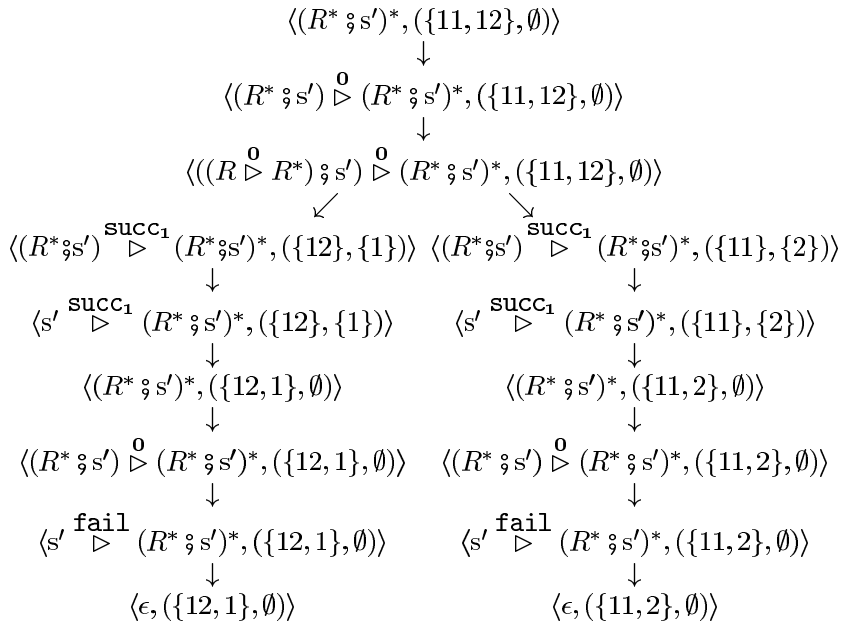
We give an example for which the i-tuple is $(\mathtt{fail}_1 = \mathtt{fail}_2 = \mathtt{sync} = \mathtt{if}) < (\mathtt{succ}_1 = \mathtt{succ}_2 = \mathtt{cond})$. Imagine that our primitive function is the following, and that our initial multiset is: $\{11, 12\}$.

$$\begin{array}{lcl} R & \stackrel{\text{def}}{=} & (B, A, \mathcal{S}_{SC1}) \\ \text{where} & & \\ B((x),(y)) & = & (x > 10) \wedge (y > 10) \\ A((x),(y)) & = & \{x - 10\} \end{array}$$

Now, we should *expect* the following, because 11 can be used as context for 12 and *vice versa*.

$$\langle (R^* \mathbin{;} \mathrm{s}')*, (\{11, 12\}, \emptyset)\rangle \Longrightarrow^*_{PT} \langle \epsilon, (\{1, 2\}, \emptyset)\rangle$$

However, this simple combination of the encodings for context and for synchronisation yields the following derivation tree (some steps are omitted):



But now, neither can 12 be rewritten to 2 nor can 11 be rewritten to 1, so it is impossible for us to obtain the desired result. Our difficulty arises because of the interaction between synchronisation and data context. When we rewrite a number of elements of the state logically in parallel, we want those elements to remain in the state as data context, in case they are needed for other rewrites before a synchronisation occurs. We wish, therefore, that the data context be initialised when a state synchronisation occurs and be left untouched between state synchronisations. We therefore suggest the following two selection functions:

$$\mathcal{S}_{G1}(B, A, (\sigma, \rho, \tau)) \stackrel{\text{def}}{=} \begin{cases} (\mathtt{succ}_1, \mathtt{succ}_2, (\sigma - \vec{\sigma_1}, \rho, \tau \uplus A(\vec{\sigma_1}, \vec{\rho_1}))) \\ \qquad \text{if } \exists \vec{\sigma_1} \subseteq \sigma, \exists \vec{\rho_1} \subseteq (\rho - \vec{\sigma_1}).B(\vec{\sigma_1}, \vec{\rho_1}) \\ (\mathtt{fail}_1, \mathtt{fail}_2, (\sigma, \rho, \tau)) \qquad \text{otherwise} \end{cases}$$

$$\mathcal{S}_{G2}(B, A, (\sigma, \rho, \tau)) \stackrel{\text{def}}{=} (\mathtt{fail}_3, \mathtt{succ}_3, (\sigma \uplus \tau, \sigma \uplus \tau, \emptyset))$$

Now our state is a triple of multisets. The first element is the old state, the second is the data context and the third is the new state which is currently under construction. Note that any context examined cannot overlap with the elements which will be removed from the multiset (as $\exists \vec{\rho_1} \subseteq (\rho - \vec{\sigma_1})$). With our state being a triple of multisets, we have to redefine our state synchronisation function to correspond to the new form of the state. For any B, A;

$$! \stackrel{\text{def}}{=} (B, A, \mathcal{S}_{G2})$$

We define:

$$\begin{array}{rcl} i_G & \stackrel{\text{def}}{=} & (\mathtt{fail}_1 = \mathtt{fail}_2 = \mathtt{fail}_3 = \mathtt{sync} = \mathtt{if} = \mathtt{semi}) \\ & < & (\mathtt{succ}_1 = \mathtt{succ}_2 = \mathtt{succ}_3 = \mathtt{cond}) \end{array}$$

Now we get the behaviours that we want, as demonstrated in Figure 3.5. Our context elements stay unchanged until a state synchronisation takes place. This means that, even if a particular element has been removed from the argument multiset, its 'shadow' remains behind it in the data context multiset. Therefore, a number of data context-sensitive reductions followed by a synchronisation really do behave as if they were in a truly parallel composition, rather than performed in an interleaved fashion. Furthermore, a primitive function which is applied a number of times to the multiset before a state synchronisation occurs is reduced, in effect, in parallel with itself.

Making this last explicit clarifies an issue which seems to this author to be somewhat muddied in the Γ literature. The ability of functions to reduce in parallel (with themselves) has been claimed to be a property of Γ, but was never explicitly stated in the semantics, which describes an *interleaving* composition of (different) functions but only allows a primitive function to be applied once at each step. This in itself is a perfectly acceptable definition of interleaving, which could be used to prove that, for example, functions can be applied in parallel with themselves. Unfortunately, the argument is not so constructed, leading to initial confusion among (this member of) the audience. Consider the following statement, from [19, page 4]. Essentially the same quote appears in [65, page 343]. The discussion concerns a definition of a primitive function (R, A), which determines the maximal element of a multiset.

> Nothing is said in this definition about the order of evaluation of the comparisons; if several disjoint pairs of elements satisfy property R, the comparisons and replacements can even be done in parallel.

This is true, in Γ, but is not explicit in the semantics. In fact, partitioning the multiset and applying two copies of the same function to disjoint subsets of the multiset appears to be a *de facto* violation of the rule for primitive function application, which applies a single primitive function atomically to the whole multiset (although the function might only be able to examine a part of it). The property in question can, in fact, be proved, but the fact that it needs to be proved is not stated.

To understand the practical applications of our new approach, consider a one-dimensional cellular automaton (CA) with neighbourhood one. CAs are explained in Chapter 1 and in [145]. At every reduction step, all elements of the array are rewritten. The new value of each array element is calculated from the current value of that element and of the old cells' neighbours. Figure 3.6 shows a possible execution of a one-dimensional CA in schematic form. Figure 3.7 shows the execution of a CA, executing with Wolfram's rule 90 [145, page 17]. Clearly, the same production rule can be applied

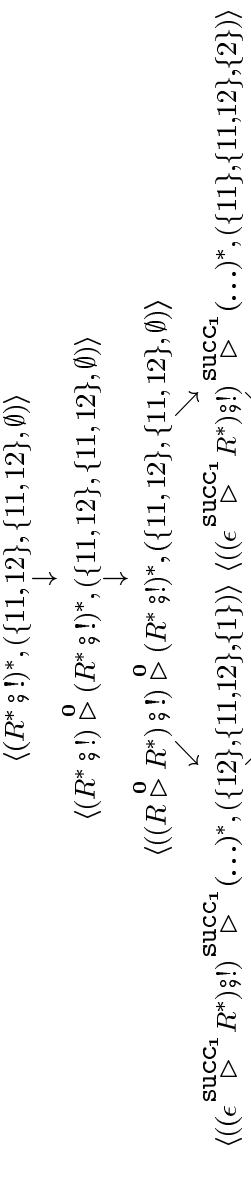


Figure 3.5: Logically parallel, data context-sensitive reductions.

many different times to different elements during a single iteration of the automaton (i.e. before a state synchronisation occurs). Therefore, functions (the production rules) are effectively being applied logically in parallel with themselves and with each other.

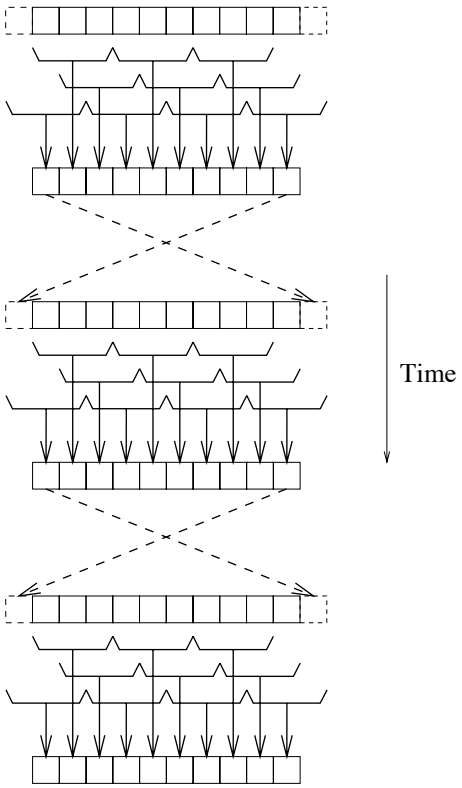


Figure 3.6: Schematic of the execution of a one dimensional cellular automaton with neighbourhood one. At every step, all array elements are rewritten. A rewrite consists of calculating a new value for each array element on the basis of its own value and those of its immediate neighbours. Rewrites are shown as arrows, whose upper half encompasses both the element to be rewritten and its data context. Dotted lines indicate copying elements for use as context for the boundary elements of the CA: boundaries may be cyclic, as here, to preserve the energy of the system. Note that each generation of the CA is rewritten data context-sensitively and logically in parallel.

Another example of context-sensitive, synchronised reduction is that of Lindenmayer systems [91], which are explained at length in Chapter 1. Suffice to give an example of their execution, in Figure 3.8.

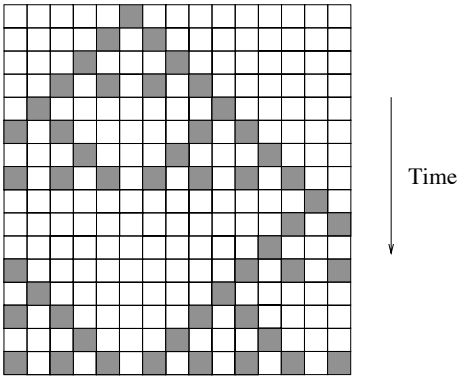


Figure 3.7: Part of an example execution of a one dimensional cellular automaton with neighbourhood one. In this example, non-cyclic boundary conditions are used: the grid is infinite, and only a part of it is seen here.

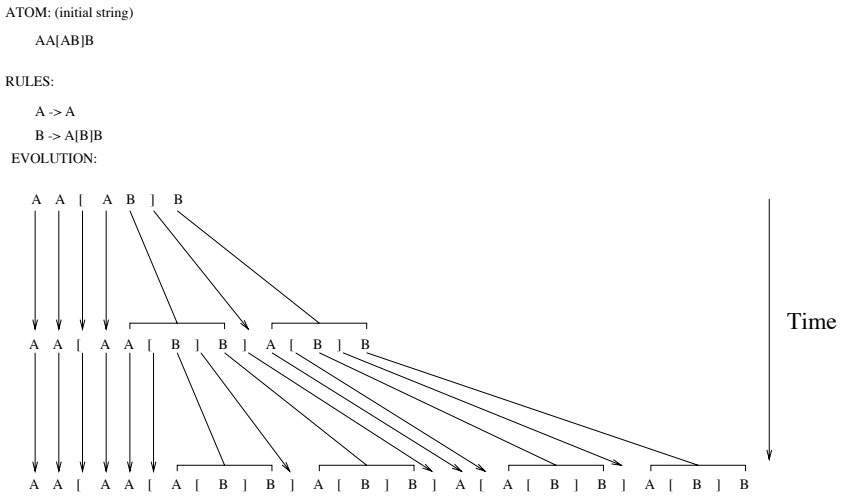


Figure 3.8: Execution of a bracketed IL-system[118, 115]. At each step, every element of the string is updated logically simultaneously. using some rule applicable to that element. To those elements to which no rule applies, the identity function is applied.

3.6 Encoding Γ in PT with $\mathcal{S}_{G1}$ and $\mathcal{S}_{G2}$.

The extra freedom we have offered the programmer in terms of when to synchronise delivers insight into the properties of Γ's interleaving composition and its relationship with a true parallel operator. Although we successfully

gave an encoding into PT of Γ in Section 2.3.1 of Chapter 2, we can now give another encoding, which makes explicit the state synchronisation step after every Γ primitive function application. Consider the following translation of Γ into PT:

$$\begin{array}{lll} [\![(B,A)]\!] & \stackrel{\text{def}}{=} & ((B,A,\mathcal{S}_{G1}) \fatsemi !)^* \\ [\![P+Q]\!] & \stackrel{\text{def}}{=} & [\![P]\!] \,|||\, [\![Q]\!] \\ [\![Q \circ P]\!] & \stackrel{\text{def}}{=} & [\![P]\!] \fatsemi [\![Q]\!] \end{array}$$

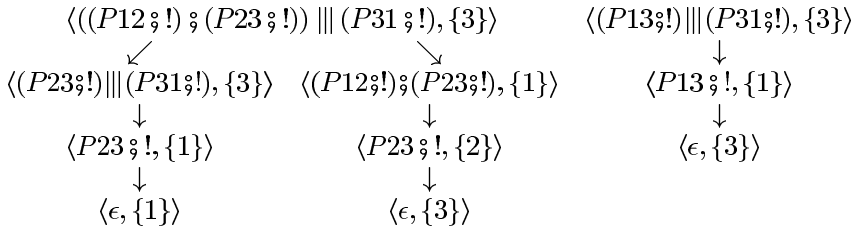
where:

$$\begin{array}{lll} i_\Gamma & \stackrel{\text{def}}{=} & (\mathbf{0} = \texttt{if} = \texttt{fail}) \\ & < & \texttt{sync} \\ & \leq & (\texttt{semi} = \texttt{cond} = \texttt{succ}_1 = \texttt{succ}_2 = \texttt{succ}_3) \end{array}$$

Proposition 9 (Correctness of the translation) *Our translation of* Γ *into PT is correct.*

Proof by induction over the structure of the semantics. The proof can be seen in Appendix A.3.

We give the possible reduction paths for each of two Γ programs, which were encoded according to our scheme. The possible reduction paths mimic those of [96], thereby showing that the context-sensitive behaviour of Γ is preserved:



3.7 Addendum: the MSTL Goblin

In the light of the investigations of program and data context-sensitivity, failure synchronisation and state synchronisation undertaken in the last two chapters, we present a new MSTL, Goblin. Goblin allows data contexts to be specified for primitive functions and makes state synchronisations explicit. Goblin's interleaving operators are program context-insensitive, in the sense that inapplicable primitive functions are just as interesting as applicable primitive functions. Goblin therefore suffers from none of the compositionality problems discussed in Chapter 2. This seems to this author to make Goblin preferable to both Γ and to CGP.

The concrete syntax of Goblin is presented in Figure 3.9. Goblin's operational semantics is defined by its translation into PT, given in Figure 3.10.

```
system    =  { typedec}* program multiset defs

typedec   =  type '==' { type name ','}* type name ';'    (type declaration)

name      =string

type      =string

prog.     ='!'                                    (synchronise)
          | 'empty'                               (empty program)
          | identifier                            (primitive function)
          | 'loop' n 'on' prog.                   (loop n times on prog.)
          | '(' prog. ')'                         (brackets)
          | prog. ';' prog.                       (sequential composition)
          | prog. '|' prog.                       (choice)
          | prog. '|||' prog.                     (interleaving composition)
          | 'if' prog. 'then' prog. 'else' prog.  (conditional execution)
          | 'when prog. 'do prog.                 (conditional execution)
          | 'unless' prog. 'do' prog.             (conditional execution)
          | prog. '*'                             (repetition)
          | X '=' prog.                           (prog. variable binding)

defs      =function*

function  =identifier '=' '(' arg* ')' [cases] ';'   (synchronising prim. fn.)
          | identifier '=' '[' arg* ']' [cases] ';'  (non-synchronising prim. fn.)

arg       =type tuple                             (remove multiset element)
          | '?' type tuple                        (read multiset element)

cases     ={ '→' [predicate ':'] multiset }+

multiset  ='{' [ { tuple ',' }+ tuple ] '}'

tuple     =element
          | '(' { element  ',' }* element ')'

condition=expression

element   =integer
          | float
          | boolean
          | string
          | variable
          | variable '.' name                     (tuple element projection)
```

Figure 3.9: The concrete syntax of Goblin. Expressions are defined standardly. 'Prog.' is an abbreviation for 'program'

$$i_{\text{Goblin}} \stackrel{\text{def}}{=} (\texttt{semi} = \texttt{fail}_1 = \texttt{fail}_2 = \texttt{sync}) < (\texttt{succ}_1 = \texttt{succ}_2 = \texttt{succ}_3 = \texttt{cond})$$

$$\begin{array}{lcll}
[\![(P)]\!] & \stackrel{\text{def}}{=} & [\![P]\!] & \\
[\![!]\!] & \stackrel{\text{def}}{=} & (\text{True}, \emptyset, \mathcal{S}_{G2}) & \\
[\![P;Q]\!] & \stackrel{\text{def}}{=} & [\![P]\!] \fatsemi [\![Q]\!] & \\
[\![P \mid Q]\!] & \stackrel{\text{def}}{=} & ([\![P]\!] \fatsemi [\![Q]\!]) \mid ([\![Q]\!] \fatsemi [\![P]\!]) & \\
[\![\texttt{if } P \texttt{ then } Q \texttt{ else } R]\!] & \stackrel{\text{def}}{=} & [\![P]\!] \stackrel{\mathbf{0}}{\triangleright} [\![Q]\!] : [\![R]\!] & \\
[\![\texttt{when } P \texttt{ do } Q]\!] & \stackrel{\text{def}}{=} & [\![P]\!] \stackrel{\mathbf{0}}{\triangleright} [\![Q]\!] : \epsilon & \\
[\![\texttt{unless } P \texttt{ do } Q]\!] & \stackrel{\text{def}}{=} & [\![P]\!] \stackrel{\mathbf{0}}{\triangleright} \epsilon : [\![Q]\!] & \\
[\![\texttt{loop } n \texttt{ on } P]\!] & \stackrel{\text{def}}{=} & [\![P]\!] \fatsemi [\![\texttt{loop } n-1 \texttt{ on } P]\!] & \text{if n} > 0 \\
[\![\texttt{loop } n \texttt{ on } P]\!] & \stackrel{\text{def}}{=} & \epsilon & \text{if n} = 0 \\
[\![P \mid\mid\mid Q]\!] & \stackrel{\text{def}}{=} & [\![P]\!] \mid\mid\mid [\![Q]\!] & \\
[\![X = P]\!] & \stackrel{\text{def}}{=} & X = [\![P]\!] & \\
[\![P*]\!] & \stackrel{\text{def}}{=} & \mu X.[\![P]\!] \stackrel{\mathbf{0}}{\triangleright} X : \epsilon & \\
[\![\texttt{`(' args `)' cases}]\!] & \stackrel{\text{def}}{=} & [\![\texttt{args cases}]\!]_{pf} \fatsemi ! & \\
[\![\texttt{`[' args `]' cases}]\!] & \stackrel{\text{def}}{=} & [\![\texttt{args cases}]\!]_{pf} & \\
[\![\texttt{args cases}]\!]_{pf} & \stackrel{\text{def}}{=} & (B(\vec{x},\vec{y}), A(\vec{x},\vec{y}), \mathcal{S}_{G1}) & \\
\text{where} & & & \\
\vec{x} & = & [\![\texttt{args}]\!]_{ra} & \\
\vec{y} & = & [\![\texttt{args}]\!]_{nra} & \\
B(\vec{x},\vec{y}) & = & [\![\texttt{cases}]\!]_{b} & \\
A(\vec{x},\vec{y}) & = & [\![\texttt{cases}]\!]_{a} &
\end{array}$$

Figure 3.10: Goblin's translation into PT. Selection functions $\mathcal{S}_{G1}$ and $\mathcal{S}_{G2}$ are to be found in Section 3.5.

Our translation function is simplified by the assumption that primitive functions are inlined into the main body of code before being translated into PT.

The translation of all constructs except the primitive functions is straightforward. Primitive functions (which have the form of case statements in Goblin) are translated in several steps. The function arguments (both non-data contextual and data contextual) are collected together into two sequences, the non-data contextual arguments by $[\![]\!]_{ra}$ and the data contextual by $[\![]\!]_{nra}$. The condition function B is generated by $[\![]\!]_b$, which gives the disjunction of all the conditions occurring in case statements. The action function A is also a conditional, generated by $[\![]\!]_a$, returning different result multisets for each case in the primitive function. We omit some of the technical details of the translation of primitive functions: the details only concern collecting the arguments into sequences and organising the conditions. Furthermore, Goblin features multiple multisets: every element has associated with it a multiset name (a *chromatic type* in the terminology of [95]). To avoid the tedious machinery associated with explicitly checking the chromatic type of every element, we assume that all elements passed to B and A are of the correct chromatic types. In [95], we show how to infer chromatic types.

We give below an example Goblin program (written in the concrete syntax), together with its translation. Our example program is:

```
max* { ... }

where
max = (Int x, ?Int y)-> x < y: {};
```

The translation of this program into PT is as follows:

$$
\begin{array}{ll}
[\![\texttt{max*}]\!] & = \mu X.[\![\texttt{max}]\!]_{pf} \overset{0}{\triangleright}: \epsilon \\
[\![\texttt{(Int x, ?Int y)-> x < y:\{\}}]\!] & = [\![\texttt{(Int x, ?Int y)-> x < y:\{\}}]\!]_{pf} \fatsemi ! \\
[\![\texttt{(Int x, ?Int y)-> x < y:\{\}}]\!]_{pf} & = (B(\vec{x},\vec{y}), A(\vec{x},\vec{y}), \mathcal{S}_{G1}) \\
\vec{x} & = [\![\texttt{Int x, ?Int y}]\!]_{ra} = \text{Int } x \\
\vec{y} & = [\![\texttt{Int x, ?Int y}]\!]_{nra} = \text{Int } y \\
B(\vec{x},\vec{y}) & = [\![\texttt{-> x < y: \ \{\}}]\!]_b = x > y \\
A(\vec{x},\vec{y}) & = [\![\texttt{-> x < y: \ \{\}}]\!]_a = \text{if } x > y \text{ then } \emptyset
\end{array}
$$

Therefore, we have:

$$
\begin{array}{l}
\mu X.((B(\text{Int } x, \text{Int } y), A(\text{Int } x, \text{Int } y), \mathcal{S}_{G1}) \fatsemi !) \overset{0}{\triangleright} X : \epsilon \\
\text{where} \\
\qquad B(\text{Int } x, \text{Int } y) = x > y \\
\qquad A(\text{Int } x, \text{Int } y) = \text{if } (x > y) \text{ then } \emptyset
\end{array}
$$

We give an intuitive description of the dynamic semantics of Goblin, for each construct in the concrete syntax. Application of a primitive function to a multiset results in replacement of the function's arguments by its results iff the predicate is satisfied. There are two kinds of primitive function: those which immediately synchronise the state and those which do not. State synchronising primitive functions update the state as soon as they are applied.

Non-synchronising primitive functions synchronise the state either (i) after application of the next synchronising primitive function or (ii) at the next '`!`' function. '`!`' forces a state synchronisation. If a function makes use of a data-context element `e` from multiset `M`, this is indicated by `?M e`. The loop construct `loop n on P` attempts to apply `P n` times. Sequential composition of programs is standard. Non-deterministic reduction `P | Q` reduces `P` and `Q` in any order. An interleaving composition of two programs `P ||| Q` proceeds non-deterministically. At each reduction step, it selects either `P` or `Q` and reduces that program by one step. To avoid excessive use of empty programs, conditional function application can be written in three ways. `if P then Q else R` is the full form. `when P do Q` is equivalent to `if P then Q else empty`. Lastly, `unless P do Q` is equivalent to `if P then empty else Q`. Finally, `P*` applies `P` to a multiset as many times as possible. `X = P` names a program fragment, allowing recursive programs to be defined in terms of themselves. Termination occurs when the program is empty. We see a number of examples of Goblin programs in Chapter 4.

3.8 Future work

We mentioned in Section 3.2 the possibility that the possible parallelism available to a multiset program which makes use of data context might be higher than that of a program which does not make use of data context. We should investigate this issue: if it turns out that our intuition is correct, it is another strong motivation for including data context in future multiset transformation languages.

Our current approach to synchronisation demands that the programmer explicitly expresses when a state synchronisation is to occur. Wim Vree has suggested clarifying whether or not the explicit state synchronisation can be omitted, if a more complex i-tuple is used and the interleaving operator deal with its own synchronisation implicitly. Unfortunately, there was not time to examine this very interesting suggestion.

There seems to be a relationship between the operators of Chaudron's Schedules [36] and those of PT (especially when a non-prescriptive parallel operator is described). So far, however, we have not investigated the relationship between Schedules and PT with state synchronisation and with or without data context. This would be an interesting topic to explore.

Finally, the multisets that have been used as example states in this chapter are passive in the sense that they do not contain tuples which are themselves programs. Given the current interest in using multiset transformation languages as a basis for coordinating programs or applications [1], the current approach should be extended to include *programs* as possible multiset elements. The precise technical impact of such a change should be assessed.

3.9 Conclusions

We have demonstrated how to encode state synchronisation and data contexts into PT using new selection functions. We have discussed the relationship between their combination and true parallel operators (both prescriptive and non-prescriptive). We have shown that interleaving, when augmented with data contexts, does not give the same possible behaviours as a truly parallel operator, at least when primitive functions are reduced atomically.

The work presented in this chapter is orthogonal to that discussed in Chapter 2: we make no assumptions here about the the i-tuple's elements, except when presenting particular examples. This means that it is possible to generate logically parallel, data-context sensitive versions of all the languages explored in Chapter 2, with the appropriate choice of i-tuple.

Our original motivation for investigating the constructs explored in this chapter was to show how to encode L-systems [91] (both OL-systems and IL-systems) into PT. Rewriting a string in L-systems is carried out in a lock-step fashion, with all symbols in the current string being rewritten logically simultaneously. Furthermore, the presence of data contexts in IL-systems prompted the investigations of the interactions between context and parallelism, in large part prompted by Professor Prusinkiewicz' criticisms of the 'no overlapping sub-states' policy common in the semantics community when defining parallel operators [114]. Unfortunately, the multiset was and remains unstructured, making it awkward to encode data structures. Therefore, our attention has been limited in this chapter to the control-flow issues attending translations from an L-system-like parallel reduction theory to a multiset transformation language. Our example programs will have to wait.

Chapter 4

Modelling stochastic phenomena using multiset transformation

We propose using multiset transformation languages as a general computational model for simulations of stochastic phenomena. Multiset transformation languages are general enough to capture a large number of stochastic effects and possess a well-defined mathematical meaning. We show that, despite initial appearances, the non-determinism present in multiset transformation languages is unrelated to the techniques required for full programmer control of general stochastic applications. Using an alternative mechanism, described in this chapter, we present multiset implementations of a number of typical stochastic applications, including Monte Carlo integral approximation, diffusion-limited aggregation, a genetic algorithm and a simulated annealing technique which realises multiple Markov chains simultaneously and therefore potentially in parallel.

4.1 Models of stochastic phenomena

When a solution or approximate solution is required to a computationally intractable physical problem, the only possible course of action is often to build a model of the physical system and to simulate explicitly the system's evolution [120]. Essential to the veracity of the results gained is the requirement that the implementation corresponds in some well-understood way to the physical system being studied. To ensure that this property holds, a model is constructed at a number of abstraction levels, starting with a mathematical description of the system being investigated (e.g. a set of partial differential equations) and ending up with a machine-executable program.

In addition to the above considerations, many physical or biological processes are stochastic in character [3]. Simulating stochastic processes on a computer requires that our computational model be able to express well-

defined stochastic effects independently of the fine operational details of the machine being used. This is particularly important to ensure repeatability of an experiment: a model has to yield (statistically) identical results when implemented in different ways on any hardware configuration.

Essential to stochastic models is the concept of random numbers and distributions thereof. Closely following Ross [120], we introduce *probability*:

Suppose that for each event A in a sample space S of mutually-exclusive events, there is a number $P(A)$. $P(A)$ is a probability iff

1. $\forall A \in S . 0 \leq P(A) \leq 1$
2. $P(S) = 1$
3. $\overset{\infty}{\underset{n=1}{\forall}} P(\bigcup_{i=1}^{n} A_i) = \sum_{j=0}^{n} P(A_j)$

Rule 1 states that the probability of a particular event happening is between 0 and 1 (inclusive). Rule 2 states that the probability that an event which happens is one of the events in the sample space is 1. Rule 3 states that, for any set of events, the probability that at least one of these events occurs is the sum of the probabilities associated with each event individually.

A random variable is a quantity of interest determined by the result of some experiment [120]. A *discrete* random variable is a random variable with at most a countable number of different possible values. A probability mass function for a discrete random variable X is defined, pointwise, as the probability $p(x)$ that each possible value x of X will occur.

Research into simulation of stochastic processes has resulted in a number of alternative models: the choice of which to use depends upon the details of the required simulation. Examples of such models include Monte Carlo methods [138], diffusion-limited aggregation (DLA) [143], genetic algorithms (GAs) and simulated annealing (SA) [83]. All of these methods have been extensively studied, allowing the simulator to verify that an implementation of their chosen model exhibits the desired properties.

The models mentioned above rely upon stochastically choosing new configurations for a system or upon determining properties of a system using stochastic methods. All of the methods are prevalent in the physical and biological modelling communities. Given that we wish to write applications taken from those communities in MSTLs, one might ask whether we *can* encode stochastic applications in MSTLs. As we show in this chapter, the answer to this question is 'Yes'. Our answer is illustrated with examples.

4.1.1 The structure of this chapter

This chapter is structured as follows. Firstly, in Section 4.2 we introduce the principles of programming MSTLs for stochastic computation. Next, in Section 4.3, we show how to control the executions of programs written in the MSTL Goblin, in three ways, using probabilities. Goblin's syntax and semantics were introduced in Chapter 3. In Section 4.4, we give (the skeletons of) a number of stochastic applications in terms of Goblin. Finally, in Section 4.5, we explain possible avenues for future work and draw our conclusions.

4.2 Stochastic versus non-deterministic computation

Arguably the most obvious choice of method for implementing stochastic choices in MSTLs is to harness their non-determinism. Indeed, this suggestion has already been made [108]. The semantics of most MSTLs possess two orthogonal sources of non-determinism.

- Non-deterministic program choice: if several applicable programs are in an interleaving composition or in a choice, the choice of which program is actually executed (first) is made non-deterministically.
- Non-deterministic element selection. When applying a primitive function, we search in the multiset(s) for a sequence of elements for which the function's predicate is satisfied. If more than one sequence satisfies the predicate, the sequence to which the function will actually be applied is chosen non-deterministically.

These two kinds of non-determinism correspond to those properties stated in Section 4.1 and can be harnessed by simple language extensions [108]. The advantage of this approach is that writing stochastic programs is directly supported by the language. Furthermore, the implementation can provide the programmer with (perhaps static) assurances that the probabilities associated with different events satisfy the laws of probability given in 4.1. An example of this approach, in a pseudo-Goblin syntax, is given below. The program takes an integer and, with a probability of 0.75, places it into multiset `Yes`. Otherwise, the integer is placed into multiset `No`.

```
f = (Int x) -> probability 0.75: { Yes x }
            -> probability 0.25: { No  x };
```

In [108], the implication is made that such an approach is *necessary* to harness probabilistic computation in an MSTL. Consider, for example, the following, from page 285 of the paper (emphasis ours):

> A useful *variant* of the Γ model is obtained by introducing probabilities for selection when one or more predicates are true for several non-disjoint subsets at the same time.

In an implementation of an MSTL intended to be used for writing stochastic applications, programmer annotations or other syntactic help *could* be added to tell the compiler which values are probabilities. Knowing which values are probabilities can help the runtime system ensure that the laws of probability, given in Section 4.1, are satisfied throughout the program's execution. However, while such a suggestion may offer the programmer considerable assistance, it is not a necessary condition for implementing stochastic applications in an MSTL. In other words, we can encode stochastic applications in an MSTL *without* having to add any machinery to the language. We demonstrate our claim in the following sections, by giving a number of

examples of stochastic programs written in a representative MSTL, Goblin. All of these programs could also be written in Γ, but Γ's control-flow difficulties (discussed in Chapter 2) make the encodings awkward. We note that the control-flow difficulties from which Γ suffers are unrelated to the stochastic extensions to Γ suggested in [108].

4.3 Stochastic MST program reduction

We show how to implement stochastic programs in MSTLs by making it possible to select both functions and multiset elements probabilistically. The exact stochastic properties of the selection process depend on two factors:

1. The probability mass function of the random number generator.
2. The relative sizes of the intervals spanned by the bounds of each element.

There are two ways in which the probabilities associated with a particular choice can be assigned. Either they can be assigned statically, so that the probabilities associated with each possible choice are written into the program, or they can be assigned dynamically, so that the probabilities associated with each possible choice are determined at run-time. In the interests of thoroughness, we give an example of each of these possibilities, in the sections shown in the table below. All of our programs are written in Goblin, whose syntax and semantics are given at the end of Chapter 3. All the primitive functions used in these examples are synchronising: the state is updated as soon as the function is applied. Synchronous (and non-synchronous) primitive functions are also described in Chapter 3.

	Function choice	Element selection	Termination
Statically-determined probabilities	§4.3.2	§4.3.4	§4.3.6
Dynamically-determined probabilities	§4.3.3	§4.3.5	§4.3.7

4.3.1 Random number generators

Random number generators are trivial to write in Goblin. This one uses the linear congruential method of Lehmer [90, 84]. b and m are constants. `seed` is the seed.

```
Random == real value;

random* { Random seed }

where random = (Random x) -> Random (((x.value * b) + 1) % m);
```

Each application of function `random` removes the current value of multiset `Random` and uses it as a seed to generate the next pseudo-random number. This new number is placed back into multiset `Random`. Throughout the rest of this paper, we omit the details of random number generation from our examples.

4.3.2 Functions chosen with probabilities determined statically

We show a program which applies a function, chosen at random. The probability of a particular function being chosen is determined statically.

```
Random == float value;

random; (f1 ||| f2) { Random number, ... }

where
f1 = (?Random r,...) -> (0.0 <= r.value < 0.3 and ...):{...};
f2 = (?Random r,...) -> (0.3 <= r.value < 1.0 and ...):{...};
```

Multiset `Random` contains a single random number in the interval $[0, 1)$. Functions `f1` and `f2` take that random number as an argument. Each function requires that the value of that random number be between certain bounds. Assuming that the random numbers are uniformly distributed in $[0, 1)$, the probability that function `f1` (respectively `f2`) is chosen is 0.3 (respectively 0.7).

4.3.3 Functions chosen with probabilities determined dynamically

We wish to associate probabilities with function choices such that the probability of a particular function being picked can alter as the program's reduction progresses. To do so, we associate a tagged multiset with every function (multiset `F1` with function `f1` etc.). Each of these multisets contains a single tuple consisting of a pair of bounds. The bounds indicate how likely each function is to be applied. By representing the bounds on functions using multiset elements, we can alter the bounds during the program's reduction.

```
Random == float value;
F1     == float upper, float lower;
F2     == float upper, float lower;

random; (f1 ||| f2) { Random seed, ...}

where
f1 = (?Random r, F1 f, ...)
  -> (f.lower <= r.value < f.upper): { F1 f', ... }
  ->                                 { F1 f'', ... };

f2 = (?Random r, F2 f, ...)
  -> (f.lower <= r.value < f.upper): { F2 f', ... }
  ->                                 { F2 f'', ... };
```

Which function is chosen depends both upon the random number generated and the upper and lower bounds associated with each function. If the random number falls between the upper and lower bounds for function `f`n, then function `f`n can be chosen. Whether the function can or cannot be applied, the bounds in multiset `F`n can be altered, changing the probability that the function is picked in the future. It is up to the programmer to ensure that the bounds for every function are non-overlapping. The general problem of ensuring that the sum of the probabilities is unity is discussed in section 4.4.3.

4.3.4 Subset selection with probabilities determined statically

We show how to encode stochastic programs such that the probabilities of particular elements being chosen are determined statically. We separate our treatment into two parts, treating selection of single elements separately from selection of non-singleton subsets. The reason is that generating probabilities for non-singleton subsets is more difficult: we have to be able to associate a single probability with a number of multiset elements. In the case of singleton subsets of the multiset, we have but to assign a probability to each element of the multiset singly.

Single element selection with probabilities determined statically

To select a single element of the multiset probabilistically, we can generate a random number and a multiset element and check the values of both. This will ensure that, with a particular probability, an element with a particular value will be chosen. An example program is given below, wherein the choice is performed by function `grab`. The program selects a single integer from the multiset. There is a probability of 0.7 that the chosen number will be smaller than 5 and a probability of 0.3 that it will be larger.

```
Random == float value;
Item   == integer value;
Chosen == float value;

random; grab { Random seed, Item item, ... }

where
grab = (Item i, ?Random r)
  -> (0.0 <= r.value < 0.7) and (i.value <  5): { Chosen i };
  -> (0.7 <= r.value < 1.0) and (i.value >= 5): { Chosen i };
```

Non-singleton subset selection with probabilities determined statically

By modifying the above program to select a number of multiset elements together, we can choose non-singleton subsets of the multiset stochastically. We give an example, below. The program has a probability of 0.7 of choosing two unequal integers and a probability of 0.3 of choosing two equal integers.

```
Random == float val;
Int    == int val;
Chosen == int fst, int snd;

random; grab { Random seed, Int someint, ... }

where grab = (?Random r, Int x, Int y)
 -> (0.0 <= r.val < 0.7) and (x != y):{Chosen (x.val, y.val)}
 -> (0.7 <= r.val < 1.0) and (x == y):{Chosen (x.val, y.val)};
```

After the program has been reduced, multiset `Chosen` contains a pair of elements whose values fall into the desired range. The problem of maintaining at unity the sum of the probabilities associated with all possible choices, is addressed in Section 4.4.3.

4.3.5 Dynamic variation of probabilities for subset selection

We show how to encode stochastic programs such that the probabilities of particular elements being chosen are determined at runtime. As in Section 4.3.4, we separate our treatment into two parts, treating selection of single elements separately from selection of non-singleton subsets.

Single element selection with dynamically-determined probabilities

To select a single element of the multiset probabilistically, we can associate bounds with each element and generate a random number. As in Section 4.3.3, if the random number falls between the bounds of an element, then that element is picked by function `grab`. An example program is given below:

```
Random == float value;
Item   == float value, float lower, float upper;
Chosen == float value;

random; grab { Random seed, Item item, ... }

where
grab = (Item i, ?Random r)
  -> (i.lower <= r.value < i.upper): { Chosen i };
```

Non-singleton subset selection with dynamically-determined probabilities

To choose probabilistically a non-singleton subset of the multiset, we need to select several elements, on the basis of their values only. If we wish to use the bounds technique previously introduced, we can use a pair of functions which calculate the probability bounds associated with a number of elements. For example, we could use a function `prob_lo()` to calculate the lower bound associated with a number of elements and `prob_up()` to calculate the upper

bound. This approach enables us to tie particular probability ranges to the values of a number of elements: if a generated random number falls between those bounds, then those elements are picked. As an example, consider the following program, which removes pairs of elements from the multiset with dynamically-determined probabilities:

```
Random == float value;
Int    == int val;
Chosen == int fst, int snd;

random; grab { Random seed, Int someint, Int otherint, ... }

where grab = (?Random r, Int x, Int y)
 ->(prob_lo(x.val, y.val) <= r.value < prob_up(x.val, y.val)):
   { Chosen (x.val, y.val) };
```

After the program has been reduced, multiset `Chosen` contains a pair of elements whose values fall into the desired range. The problem of maintaining at unity the sum of the probabilities associated with all possible choices, is addressed in Section 4.4.3.

4.3.6 Termination with a statically-determined probability

We can use the same techniques as above to produce programs which terminate stochastically. The technique involves ensuring the termination of the program if an element has a particular value, which value is probabilistically assigned. In this case, the relevant element is a random number. Consider the following program:

```
Random  == float value;
Counter == int val;

(unless done do (random; increment))*

{ Counter 0, Random seed, ... }

where done      = (?Random r) -> (r.value > .5): {};
      increment = (Counter x) -> Counter (x.val + 1);
```

This program continues to apply functions `increment` and `random` unless function `done` determines that the random number is greater than one half. If so, the program terminates. The result is a random positive integer in multiset `Counter`, such that the probability of a certain integer n being reached is $\frac{1}{2^n}$.

4.3.7 Termination with a dynamically-determined probability

In this program, the probability of the program terminating decreases with each increase in the integer in multiset `Counter` until it is no longer possible

for the program to terminate. The example is somewhat artificial, but is sufficiently similar to the previous example for easy comparison.

```
Random  == float val;
Counter == int val;

(unless done do (random; increment))*

{ Counter 0, Random seed, ... }

where
done      = (?Random r, ?Counter c)->((r.val * 100)>(1/c.val)):;
increment = (Counter x) -> Counter x.val + 1;
```

4.4 Example stochastic applications

Now that we have shown the ways in which a program's reduction can be influenced by stochastic effects, we conclude with four more serious examples of probabilistic applications. These are the Monte Carlo method [138] for integral approximation, Diffusion-Limited Aggregation [143], a Genetic Algorithm (GA) [72] and Simulated Annealing (SA) [83].

4.4.1 Monte Carlo integral approximation

We use the Monte Carlo method to calculate integrals of a function. We demonstrate an approximation of π. The statement of the problem is to be found in Ross [120, pages 40-41], where the interested reader can also find more details than those given here.

Take a square of side 2 centered on the origin and a circle of radius 1, also centered on the origin (a graphical representation is given in Fig. 4.1). The area of the square is 4. The area of the circle is (πr^2 where $r = 1$) $= \pi$. Suppose that a random vector (x, y) is uniformly distributed in this square (i.e. both x and y are independent, uniformly distributed random variables). Then it follows that the probability that the point (x, y) lies in the circle is the probability that $x^2 + y^2 \leq 1$. Again, because the random variables are uniformly distributed, this equals the proportion of the square which is in the circle, which equals $\pi/4$.

Therefore, if we generate a large number of random points in the square, the proportion that will fall within the circle will be approximately $\pi/4$. As Ross shows (page 41), we can estimate π by generating a large number of pairs of random numbers (x, y) in the range $[0, 1)$ and estimating $\pi/4$ as the fraction of pairs for which $(2x - 1)^2 + (2y - 1)^2 \leq 1$.

The program to approximate π is shown below. The larger the number of iterations which are performed, the better the approximation to π:

```
Randomvec  == real x, real y;
Hits       == int hits;
PrintFloat == real value;
```

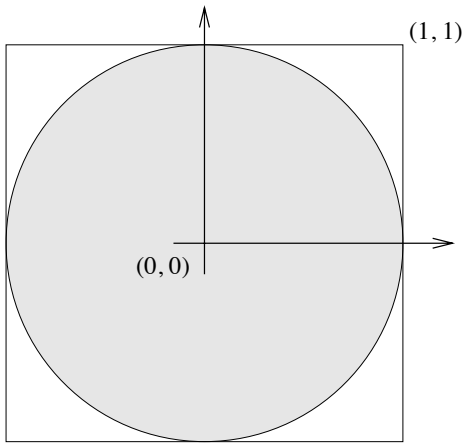


Figure 4.1: A square of side 2 containing a circle of radius 1.

```
(loop 10^9 on (randomvec; guess)); give_result

{ Randomvec (0, 0), Hits 0 }

where
randomvec = (Randomvec v)->Randomvec (random(v.x), random(v.y));

guess = (Randomvec v, Hits h)
  -> ((2* v.x) - 1)^2 + ((2 * v.y) - 1)^2 <= 1: Hits h.hits + 1
  -> Hits h.hits;

give_result = (Hits h) -> PrintFloat (4.0 * h.hits) / 10^9;
```

The program executes as follows. The loop `(randomvec; guess)` is executed 10^9 times. For each iteration, `randomvec` generates a new vector of random numbers, where `random()` contains the machinery described in Section 4.3.1. Next, function `guess` checks to see if the vector falls within the circle. If it does, then the counter in multiset `Hits` is incremented. If not, then the counter in multiset `Hits` is unchanged. This loop continues until all 10^9 guesses have been performed. Finally, function `give_result` prints the result.

4.4.2 Diffusion-limited aggregation

Diffusion-limited aggregation (DLA) was introduced in [143] and explained in Chapter 1. We give below an example Goblin implementation of DLA, wherein all primitive functions are synchronising.

```
startup;
  loop (100000) on
     (rand1; new_walker;
```

Figure 4.2: A DLA cluster with box dimension 1.64

```
(rand2; move; box; candidate; collide*; rebuild*;
(new_sides ||| move_again ||| draw))*)
```

We include only the main program of the DLA, eliding the details of the primitive functions for reasons of clarity. We show an example DLA cluster, as generated by this program, in Fig. 4.2. The program executes as follows: firstly, a new walker is placed at a random place on the birth circle (functions `rand1` and `new_walker`). Then, that walker begins to perform a random walk (functions `rand2` and `move`). If the walker reaches the death circle, it is temporarily removed by function `box`. Function `move_again` moves it back to the birth circle later on. If the walker moves into a location adjacent to the growth form, it sticks to it (functions `candidate`, `collide` and `rebuild`). In general, the birth and death radii are maintained as multiples of the diameter of the cluster. Therefore, once the current walker has accreted to the cluster, the birth and death circles are enlarged, if this is necessary, using function `new_sides`. Finally, the current figure is drawn on the screen using function `draw`.

4.4.3 A genetic algorithm: an example using normalisation

Genetic algorithms (GA) were introduced in Chapter 1. The skeleton for the genetic algorithm code is given below. All primitive functions are synchronising.

```
loop iterations on (addup*; (loop 2 on choose); crossover){...}
```

Where:

- `addup` adds up the probability associated with each possible string in the multiset.

- `choose` chooses a string from the multiset, with probability proportional to the fitness of the string.

- `crossover` takes the two strings already generated and performs a crossover operation on them.

In this example, the strings are not traversed in any particular order. For each application of `addup`, a string is chosen from the multiset and its fitness calculated and added to the total. Once the set of candidate strings for this generation is empty (i.e. has been fully processed), the computation proceeds by choosing the strings for crossover. The general problem of normalising the probabilities so that the sum of the probabilities of all the possible choices is one, is not easy. The standard implementation technique is that here used: a traverse of all the elements of the state determines the total of some quantity present in the state. Any probabilities subsequently generated are divided by this amount, thereby being normalised. The formal details of this are given in Section 1.2.7.

Notice that adding probabilistic primitives to Goblin would not help in coding this example: the probabilities of a particular choice are dependent on the fitness of particular strings relative to the total fitness of the gene pool. Thus, the dynamic changes to the multiset cause dynamic changes to both the fitness of individual strings and the fitness of the total pool. This requirement of dynamically-changing probabilities is also explicitly recognised by Krishnamurthy and Murthy [108, page 291], who, unfortunately, give no comments on the implications of this idea for their probabilistic MST model.

4.4.4 Simulated annealing

Simulated annealing (SA) [83] generates approximate solutions to combinatorial optimisation problems. The model is an abstraction of a cooling process (hence the name) in which particles explore a space for regions of low energy. Beginning with a high 'temperature', particles walk randomly through the space, moving to new locations with probabilities reflecting the particles' tendency to move to lower energy states. The initially high temperature of the particles is manifested in their ability to make large jumps in the phase space, a property which ensures that the probability is low that the particles will be trapped in non-global minima. As the particles' temperature is lowered, their tendency to settle at local minima increases, leading to exploration of progressively smaller parts of the phase space and an increase in the probability that a particle will get trapped in a local minimum. Eventually, the possible movements in the phase space which can be made by the particles are so small (due to the particles' now low temperature) that an effective freeze takes place. When the difference in entropy between two successive generations is smaller than some constant, indicating that such a

freeze has, indeed, taken place, the annealing is considered terminated. The chain of successive energies of the system is a Markov chain [120, page 181].

Implementation of SA with single Markov chain

The program below implements simulated annealing, in Goblin, with a single Markov chain. The program proceeds as follows. Firstly, the program checks to see if the difference in energy between the previous configuration and the current configuration is smaller than `small`, using function `done`. If this energy change is, indeed small, then the program terminates. Otherwise, the program continues executing. Function `update-chain` updates the current configuration (from multiset `This_Iteration`), using the current temperature (from multiset `Temperature`) and a random number (from multiset `Random`). Using these values, it generates a new configuration, placing it in multiset `New_Iteration`. Function `choose-chain` chooses between the current configuration and the new configuration. The lowest energy is adopted and placed in multiset `This_iteration`. At the same time, the current configuration is placed into multiset `Previous_Iteration`. Function `cool` cools the system every `interval` time steps. Finally, the time is increased and the whole program executed again. In what follows, $x\%y$ is the remainder of an integer division of x by y.

```
Time               == int time;
This_Iteration     == int energy, ...
New_Iteration      == int energy, ...
Previous_Iteration == int energy, ...
Temperature        == float degrees;

(unless done do (random; update-chain; choose-chain;cool;tick))*

{Time 0,This_Iteration...,Previous_Iteration...,Temperature...}

where
done = (?This_Iteration c1, ?Previous_Iteration c2) ->
  (abs(c1.energy - c2.energy) < small):;

update-chain = (?Random r, ?Temperature t, ?This_Iteration c)
  -> { New_Iteration updated(r, t, c) };

choose-chain =
(New_Iteration c1, This_iteration c2, Previous_Iteration c3)
  -> c1.energy < c2.energy:
     { This_Iteration c1, Previous_Iteration c2 }
  -> { This_Iteration c2, Previous_Iteration c2 };

cool = (Temperature t, ?Time ti)
  -> (ti % interval == 0): { Temperature newtemp(t) };

tick = (Time t) -> Time t + 1;
```

Implementation of SA with multiple Markov chains

The description given above treats Markov chain generation as an inherently sequential process. A scheme for implementing Markov Chains in parallel has been suggested in [133]. In the suggested scheme, a Markov chain is begun on a single processor. Every t time steps, all of the existing chains adopt the best (i.e. lowest energy) configuration of any of the existing chains at time t. At the same time, a new chain at the same temperature is begun on another processor. The initial configuration of the new chain is the best of all the existing chains. This scheme is represented graphically in Fig. 4.3. We implement multiple chains by having multiple elements in multisets `This_Iteration` and `Previous_Iteration`. Each of these elements is, with respect to the single-chain implementation, augmented with another field, `chain`, which indicates from which chain a particular configuration comes. We give a sketch of the multi-chain implementation of SA below.

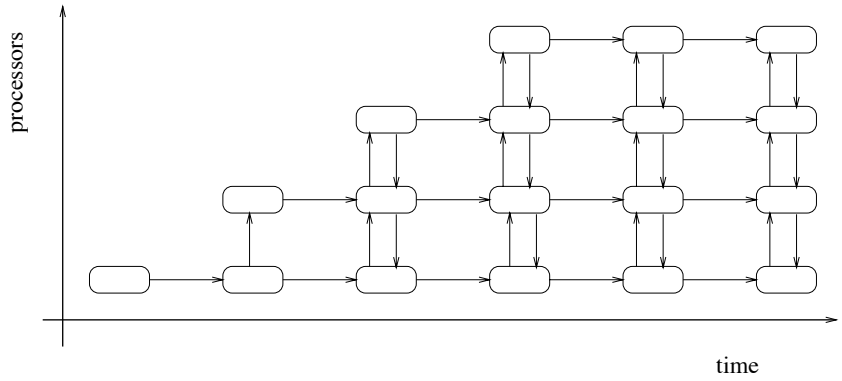


Figure 4.3: A multiprocessor implementation of parallel Markov chains. As time progresses, more processors are allocated. Each new processor begins with the best chain generated by any processor up to that time. Vertical arrows indicate communication between processors. Horizontal arrows suggest a continuity in the computation within a single processor.

```
(
  unless done do (
    (
     random; update-chain; choose-chain;
     if new-best-chain then make-new-best-chain
    )*;
    (if cool then (spawn-chain; reinitialise*); tick
  )
)*

{Time 0,
```

```
This_Iteration..., New_Iteration...,  Previous_Iteration...,
Temperature...,    Number_of_chains 1, Best_chain_so_far...}
```

The execution of this program is essentially the same as that of the single-chain version, although extra housekeeping is needed to keep track of the best chain so far, spawning new chains, and replacing all existing chains with the best chain so far. In terms of making use of probabilities, the new program is no different from the old. Function `spawn-chain` begins a new chain when the system is cooled. Function `reinitialise` replaces all current configurations with the best configuration so far.

4.5 Future work

The examples in this chapter are simple and do not attempt to be realistic models of physical processes. Instead, they should be viewed as part of a feasibility study in the usefulness of MSTLs for modeling stochastic phenomena. Despite this caveat, however, we believe that the principles which we have here demonstrated are applicable to more serious stochastic models. We also present more example stochastic programs in Chapter 5. We hope to examine more and more serious probabilistic applications in the future.

In Section 4.2, we discussed the possibility that syntactic and/or runtime help be given to the programmer to assist her in producing better stochastic programs (for example by checking that the laws of probability are satisfied for a particular event space). We have not yet explored detailed proposals in this direction.

4.6 Conclusions

Modelling many physical systems requires that stochastic simulations be performed. We claimed that ideal abstract models for this purpose, are the high-level descriptions which MSTLs offer. The details of the MSTL's implementation, together with the quirks of the underlying hardware, are hidden by a model which is compelling, easy to understand and which can be implemented in a wide variety of ways [39, 139, 70]. Murthy and Krishnamurthy [108] claimed that stochastic programming was a useful application area for MSTLs. With this we agree. However, in the same paper, Murthy and Krishnamurthy also implied that an extension of an MSTL with probabilistic constructs was necessary in order to be able easily to encode stochastic applications. This claim is false, as we have here demonstrated.

Chapter 5

Modelling environmentally-sensitive growth using multiset transformation[1]

We demonstrate the usefulness of multiset transformation for modelling growth processes which combine generational growth with environmental-sensitivity (such as photo-sensitivity and geometric constraints). Our examples are artificial structures with no immediate counterparts in physics and biology. However, our studies are intended to show the *feasibility* of multiset transformation for more realistic models. We show that multiset transformation is useful as both a formal and a computational model of growth processes.

5.1 Introduction

Modelling growth phenomena has traditionally been an important application area in formal language theory. There is a huge amount of literature available on the modelling of botanical structures using Lindenmayer systems [91]. Good overviews of the application of L-systems for modelling botanical objects can be found in [123] and [118]. An important yet problematic issue in developmental models of plants using L-systems is environmental sensitivity. Many seed plants need light for their photo-synthesis; in a realistic developmental model, this environmental influence should be included. Another environmental constraint is that collisions between parts of the growth form should be avoided: otherwise, physically impossible situations would occur. It is possible to construct generation procedures for a botanical object, for example a branching structure, which is exactly self-

[1]This chapter is based upon the paper "On modelling environmentally-sensitive growth forms and cellular automata using multiset transformation" by H. McEvoy and J. Kaandorp, which appeared in Fractals volume 4, number 4 [98].

avoiding. In reality, the influence of the environment induces irregularities in the growth process which perturbs the self-avoiding structure. For example, the growth process may be hindered by the presence of obstacles. In a realistic model of a branching structure, a collision-avoidance mechanism is a necessary prerequisite. In [117], a demonstration of environmentally sensitive L-systems is given. Examples are given in which the growth is limited by externally-defined bounding volumes. The L-systems applied to these examples are context-sensitive and the environmental influence is included by using an interpretation step on the string generated in the production rule.

A good example of an environmentally-sensitive growth model is Diffusion Limited Aggregation [143], which was described in Chapter 1. In DLA, growth is modelled using a probabilistic cellular automaton (CA). The growth form is represented by an aggregate of sites in a lattice. The probability that a site adjacent to the cluster is added to the aggregate depends on the local "nutrient" concentration. The highest probabilities occur at sites with the highest local nutrient concentration. The environmental influence is represented by the diffusion of nutrient from a source in the lattice, while nutrient is consumed by the aggregate. The DLA model serves very well as a model of a wide range of growth phenomena from physics and biology. In [74], the generative power for pattern formation of both cellular automata and L-systems is compared, where CA have the highest generative power. For modelling growth and form of more complex systems than aggregates of particles, the representation of the form by a CA remains problematic: many branching structures can more easily be described using an L-system. An alternative method is to represent the growth form by geometrical objects, the growth process by an iterative geometrical construction and to use discrete (lattice) representations to determine the environmental influence [79].

A common property of environmentally-sensitive growth is directionality. In [94], a deterministic model for directed fractal growth is discussed. Directionality in real growing systems is observed as branches tend to grow outwards from a form and not in all directions. Therefore a realistic model of such processes should allow the expression of directionality constraints.

In this chapter, we demonstrate that the growth processes of environmentally sensitive artificial branching structures and cellular automata can be modelled using the multiset transformation language (MSTL) Gamma (Γ) [16, 19, 65]. Although we use an artificial branching structure in our case studies, we intend that this chapter be seen as a feasibility study and that similar methods to those here used be applied to the more realistic morphogenetic models of physical and biological research.

In Γ, interaction between a function and a multiset only occurs when a function removes an element from the multiset or inserts an element into the multiset. In our presentation, we indicate the name of a multiset by beginning it with a capital letter. This is followed by '::' for 'remove' or ':' for 'insert'. The elements in our multisets are arbitrary tuples. These are constructed by placing a number of items, separated by commas, between

parentheses. We write `x.1` for the first element of the tuple `x`, `x.2` for the second, and so on.

5.2 Modelling Growth of Ramifying Objects

We now turn to modelling growth objects using Γ. The first example simply draws a tree, and is included because it is the basis upon which the later examples are built. The program is given in Figure 5.1. The object which it produces is shown in Figure 5.2. The multiset `Branch` contains sextuples, whose elements are, respectively, the $x-$ and $y-$ coordinates of one end of the branch, the age of the branch, the length of the branch, the direction in which the branch is growing, and the colour of the branch.

```
macro transform(x, angle, colour) =
((x.1 + (x.4 * sin(x.5))), (x.2 + (x.4 * cos(x.5))),
(x.3 + 1), (x.4 * .6), (x.5 + (angle)), (x.6 - (colour)))

macro polygon(x) = (4, x.6,
      (x.1 + (x.4*cos(x.5)*.1)),(x.2 - (x.4*sin(x.5)*.1)),
      (x.1 - (x.4*cos(x.5)*.1)),(x.2 + (x.4*sin(x.5)*.1)),
      (x.1 + (x.4*sin(x.5) - (x.4*cos(x.5) * .05))),
      (x.2 + (x.4*cos(x.5) + (x.4*sin(x.5) * .05))),
      (x.1 + (x.4*sin(x.5) + (x.4*cos(x.5) * .05))),
      (x.2 + (x.4*cos(x.5) - (x.4*sin(x.5) * .05)))

grow { Branch : (0, -.5, 0, .5, 0, 150) }

grow (Branch :: x)
  ->
  Branch : transform( x, 5.585, 2),
  Branch : transform( x, 0.175, 4),
  Branch : transform( x, 1.047, 6)
  Graphics : polygon(x)   if x.3 < 6
  -> {} otherwise;
```

Figure 5.1: A program which draws a branching object. The result of executing this program is shown in Figure 5.2. Macros `transform()` and `polygon()` define, respectively, a 6-tuple and a 10-tuple. The contents of the 6-tuple are explained in the text. The 10-tuple contains, respectively, the number of vertices in the polygon to be drawn (i.e. 4), the colour of the form, followed by four sets of (x, y) coordinates for the vertices.

The operation of the program is as follows. There are two multisets: `Branch` and `Graphics`. Elements are removed from multiset `Branch` and the conditional `x.3` $<$ `6` is evaluated with respect to the element just removed. Depending on the value of the condition, either the element is discarded (if the conditional is false) or it is *replaced* by three elements in multiset `Branch` and one in multiset `Graphics`. The side-effect of placing a tuple in multiset `Graphics` is that the tuple is used to generate a graphical image on the screen (in this case, a branch). This is how we generated all of the images

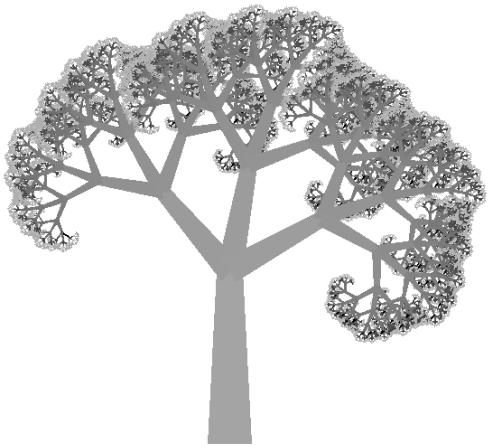


Figure 5.2: A branching object akin to broccoli. The program which generates this object is given in Figure 5.1.

Figure 5.3: The network representation of the algorithm for generating a branching structure. The algorithm is given in Figure 5.1.

in this chapter. Eventually, no more branches are left in multiset `Branch` and the computation terminates due to data starvation. Notice that the test to see whether a branch produces children is carried out locally: no global knowledge of the contents of the multiset of branches is required. In fact, the Γ model makes impossible global knowledge of the contents of an arbitrarily-large multiset: a particular function can only remove a constant-sized subset of elements of the multiset.

We can represent the Γ program as a network of multisets connected together by functions: that is, as a process network [81]. The network corresponding to the above multiset transformer is given in Figure 5.3. The rectangles represent named partitions of the multiset. The ellipses represent the functions. Data flows from the multisets, through the functions and into (perhaps different) multisets.

Notice that the representation of Γ programs as networks does not correspond to normal process networks in that message orderings are not preserved and that there is no need for a consumer to consume values as the producer produces them. Rather, the mechanism is more like that of generative communication in Linda [58]: messages are left in the multiset (tuple

space) until such time as they are retrieved.

The use of multisets rather than streams for communication between functions allows the computation to continue in a non-deterministic manner. For example, there is no need for all of the branches of a particular age to be updated before any of the next younger age is updated. This is not entirely realistic: if we model environmental factors, we would expect many of the growth points to grow at the same time, if not at the same rate. However, the Γ program can be altered to make it more realistic by including a synchronisation step after the growth of each generation of branches. This issue is discussed in greater detail in Chapter 3.

5.2.1 Environmental sensitivity 1: collisions with a fixed object

We now show the suitability of Γ programs for modelling growth of objects which are sensitive to their environment. We begin by generating a growth object whose branches cannot cross the boundaries of a box in which it is placed. The Γ program is shown in Figure 5.4. The image it produces is shown in Figure 5.5. We can represent the Γ program by a network, shown in Figure 5.6. The elements of the multisets `Line`, `Line_check` and `Line_to_draw` are sextuples with the same interpretation as in the previous example.

```
lines + check + draw

{ Line_check: (.1, .07, 0, .6, 0, 75),
  Graphics:( .55,  .05,  .55, -.55, 4),
  Graphics:( .55, -.55, -.55, -.55, 4),
  Graphics:(-.55, -.55, -.55,  .55, 4),
  Graphics:(-.55,  .55,  .55,  .55, 4) }

lines (Line:: x)
-> Line_check: transform( x, .349, 2),
   Line_check: transform( x, .300, 4),
   Line_check: transform( x, .460, 6)
     if x.5 < 9
-> {} otherwise;

check (Line_check:: x)
-> Line: x, Line_to_draw: x if not(collision(x))
-> {} otherwise;

draw (Line_to_draw:: x) -> Graphics: polygon(x);
```

Figure 5.4: A algorithm for generating a branching structure reminiscent of broccoli in a box. The result of executing the algorithm is shown in Figure 5.5. Macros `transform()` and `polygon()` are the same as before and are therefore omitted.

The function `lines` behaves in the same way as the function `grow` in

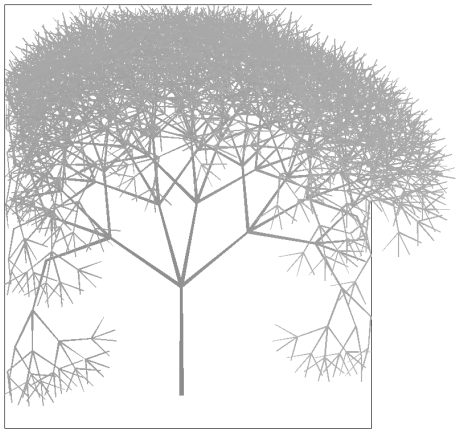


Figure 5.5: 'Broccoli in a box': a growth object generated by the program given in Figure 5.4.

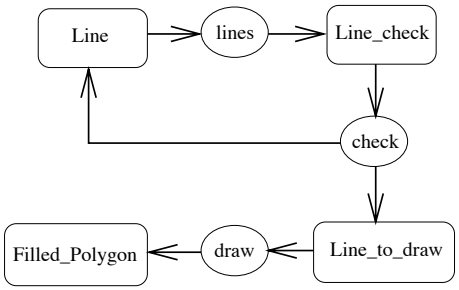


Figure 5.6: The network representation of the program for generating a boxed ramifying structure. The program is given in Figure 5.4.

the previous example, except that its results are placed in set `Line_check` instead of back into the multiset from which the arguments were removed. Function `check` takes each line in turn and checks to see whether or not it collides with the box. If it does, then it is discarded (this implies that it has no more children). If it does not collide with the edge of the box, then it is placed in multiset `Line_to_draw` to be drawn by function `draw` and a copy of it is placed back in multiset `Line` to act as the seed for the next generation of points. Again, computation is non-deterministic: it is quite possible that branches are drawn in the opposite order to that in which they were generated.

5.2.2 Environmental sensitivity 2: growth towards the light

A Γ program which draws a photo-sensitive object is shown in Figure 5.7. Figure 5.8 shows the result of executing the program. A network showing the execution of the Γ program is given in Figure 5.9.

```
macro bg = .75
macro pi = 3.14159265
macro new(x1, x2, x7, y1, y2) = \
      (((((-pi rand pi) * bg) + \
      ((1 - bg) * arctan((y1 - x1) / (y2 - x2))))) + (pi*x7))
macro neg_br(x, y) = ((y.2 - x.2) < 0)

germinate + grow + sign { Seed: (0, -.3), Sun: (1, 1) }

germinate (Seed:: x, Sun:: y)
  ->
  Pot_Branch: (x.1, x.2, 0, .3, new(x.1, x.2, 0, y.1, y.2), 60),
  /* draw sun and pot (omitted) */
  Sun: y;

sign (Pot_Branch:: x, Sun:: y)
  ->
  Signed_Branch: (x.1, x.2, x.3, x.4, x.5, x.6, neg_br(x, y)),
  Sun: y;

grow (Signed_Branch:: x, Sun:: y)
  ->
  Pot_Branch: next( x, new(x.1, x.2, x.7, y.1, y.2), 2),
  Pot_Branch: next( x, new(x.1, x.2, x.7, y.1, y.2), 4),
  Graphics: polygon(x),
  Sun: y     if x.3 < 10
  -> Graphics: polygon(x), Sun: y  otherwise;
```

Figure 5.7: A program for drawing a photo-sensitive structure. The result of executing the program is shown in Figure 5.8.

The program behaves in a similar manner to the earlier examples. This time, however, the angle of each branch to the vertical is given by a random variable with a cosine probability distribution centered on the angular difference between the current growth point and the light source. This probability distribution is adjusted to model the presence of ambient light. The level of ambient light relative to the output of the light source is given by the macro `bg` at the top of the program. Depending on this value, the tendency of the object is to grow towards the light. The lower the level of ambient light, the stronger is the tendency for the object to grow towards the light source, as one would expect.

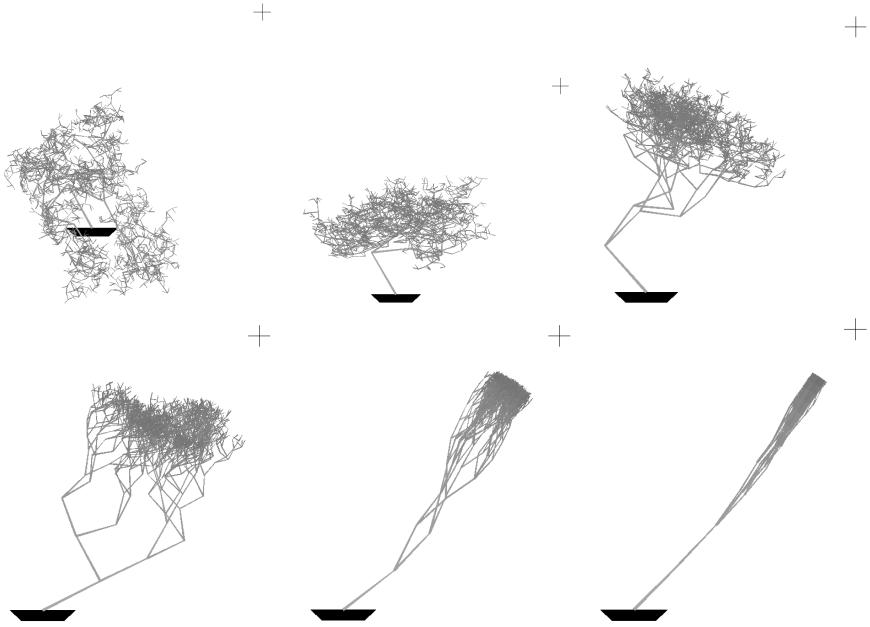


Figure 5.8: A photo-sensitive growth object. The direction of each branch segment is dependent upon the direction of the light source (a point source indicated by a cross) and the level of ambient light. The six diagrams show a growth pattern for the following levels of ambient light relative to the brightness of the light source, respectively: 95%, 75%, 55%, 35%, 15% and 5%

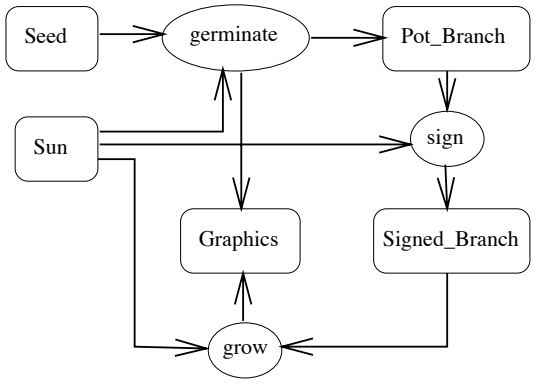


Figure 5.9: A network representing the program for generating a photo-sensitive structure. The program is to be found in Figure 5.7.

5.2.3 Environmental sensitivity 3: non-self intersection

Other interesting examples of environmentally-sensitive growth are non-self intersecting objects. The Γ program, given in Figure 5.10, has certain similarities with that given in Figure 5.1. Elements are removed from multiset `Branch` and replaced by four elements in multiset `Maybe_branch` if their generation number is smaller than six. They are discarded otherwise. Branches from this multiset are selected by function `check_start` and placed into multiset `Candidate`. Each candidate is checked for collisions with all of the branches which have already been drawn and are now stored in `Branch_check`. If a collision occurs then the candidate is discarded; otherwise, it is drawn and placed into multisets `Branch_check` and `Branch` to indicate, respectively, that it has been drawn and that it should act as a seed for the next generation of branches. The result produced by this Γ program is given in Figure 5.11. A network representation of the execution of this Γ program is given in Figure 5.12.

```
grow + check_start + check + boom + clear + draw + old2new

{ Candidate: (0, -.9, 0, .4, 0, 40) }

grow (Branch:: x)
  -> Maybe_branch : next(x, 5.323, 2),
     Maybe_branch : next(x, 6.021, 3),
     Maybe_branch : next(x, 0.175, 4),
     Maybe_branch : next(x, 0.873, 5)  if x.3 < 6
  -> {} otherwise;

check_start (Maybe_branch:: x) -> Candidate: x;

check (Candidate:: x, Branch_check:: y)
  -> Candidate: x,
     Checked_branch: y,
     Boom:1
       if collision(x, y)
  -> Candidate: x,
     Checked_branch: y
       otherwise;

boom (Candidate:: x, Boom:: b) -> {};

clear (Boom:: b) -> {};

draw (Candidate:: x)
  -> Graphics: polygon(x), Branch_check: x, Branch: x;

old2new (Checked_branch:: y) -> Branch_check: y;
```

Figure 5.10: The program to draw a non-self intersecting structure. The result of executing the program is given in Figure 5.10. Some of the housekeeping functions have been elided for the sake of clarity

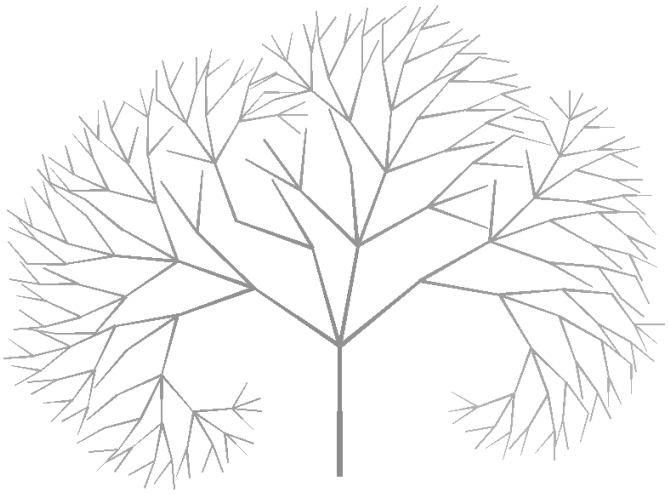


Figure 5.11: A non-self intersecting structure akin to house bamboo. The program which generates this result is shown in Figure 5.10.

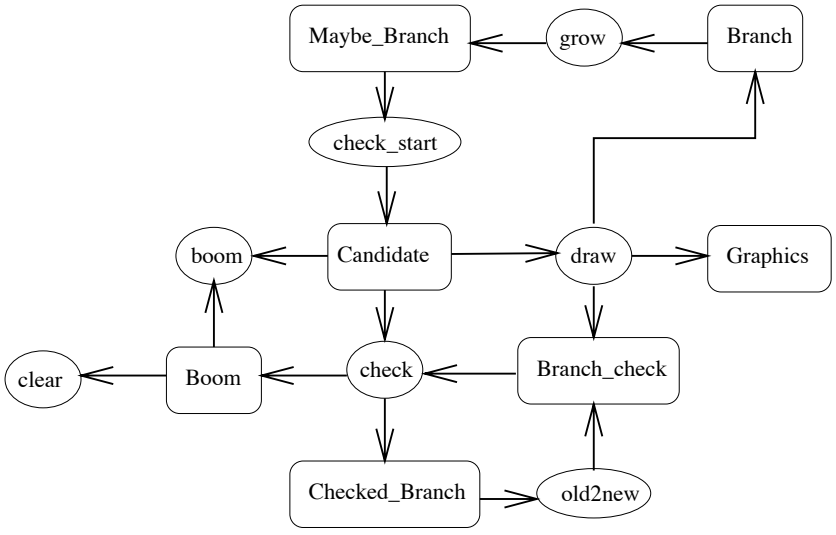


Figure 5.12: A network representing the execution of the program to draw a non-self intersecting structure. The program is given in Figure 5.10.

5.2.4 Environmental sensitivity 4: combining non-self intersection with photo-sensitivity

It is easy to combine the Γ program for photo-sensitive growth with the Γ program for non-self intersection. None of the functions in either Γ program are changed (except for the names of the multisets from which they get their input and into which they place their output). The two Γ programs are then simply composed and a number of unnecessary functions removed. The resulting Γ program is shown below, with the function definitions elided because they are unchanged. The result of running the Γ program is shown in Figure 5.13. The network interpretation of the Γ program is given in Figure 5.14.

```
germinate + sign + grow + check_start + check +
 boom + clear + draw + old2new
```

Figure 5.13: A non-self intersecting, photo-sensitive object formed through the composition of the multiset transformers for a non-self intersecting object (Figure 5.10) and for photo-sensitivity (Figure 5.7).

5.2.5 A Cellular Automaton

Cellular automata (CA) are also examples of environmentally-sensitive systems, as the state of an element's neighbours influence its evolution [74, 145]. To demonstrate the wide applicability of multiset transformation to environmentally-sensitive systems, we give an example of Γ program which generates a cellular automaton.

The skeleton for the cellular automaton program is shown below. The result of executing a 1-dimensional, five state automaton with neighbourhood 1 is shown in Figure 5.15. The network representation of the program is shown in Figure 5.16.

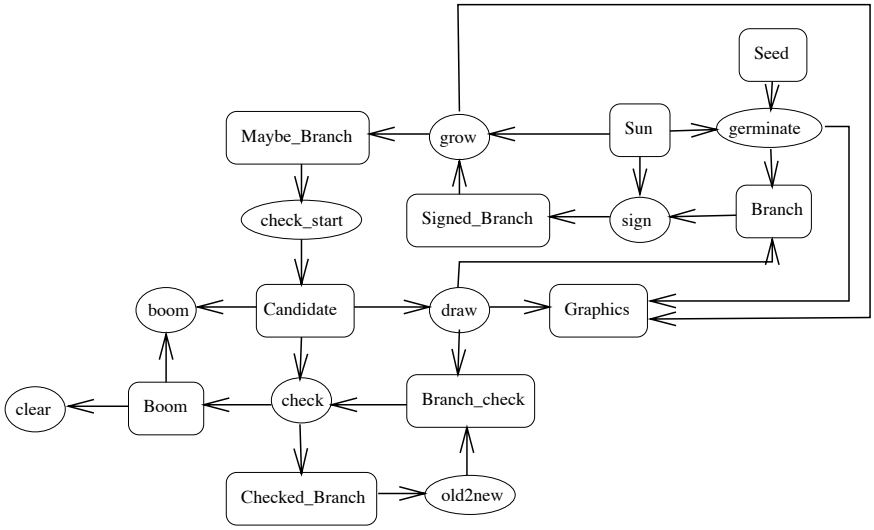


Figure 5.14: A network representing the execution of the Γ program to draw a non-self intersecting, photo-sensitive structure. The skeleton of the Γ program is given in the text.

```
center + left + right + borders + draw + next_cell +
  clear + filter + next_gen
```

The program executes as follows. Functions `center`, `left` and `right` pick, respectively, the element to be rewritten and its left and right neighbours. If the current cell is at the edge of the array (and therefore has no left or no right neighbour), function `borders` adds the appropriate 'border' element. Function `next_cell` generates the new value of the given cell, while functions `clear` and `filter` ensure that all the elements of the current generation of the automaton are rewritten before function `next_gen` initialises calculation of the next generation of the automaton. Function `draw` draws each cell on the screen.

5.3 Conclusions

We have shown the advantages of generative Γ programs for modelling environmentally-sensitive growth objects. We have demonstrated that environmentally sensitive growth processes can easily be modelled. Although we used a simple and artificial example of a ramifying object, the model of the influence of the physical environment is basically realistic. In particular, the results shown in Figure 5.8 are a foundation upon which more sophisticated models could be built. We have shown that it is possible to combine, in a single Γ program, the effects of the influence of light on the growth process with the geometric constraints imposed by the presence of obstacles.

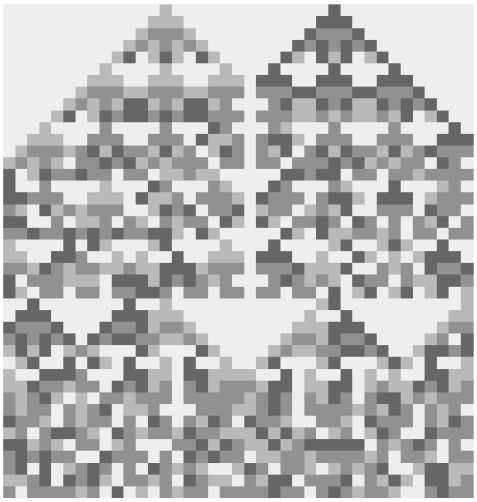


Figure 5.15: Execution of a 1-dimensional, five-state cellular automata with neighbourhood 1.

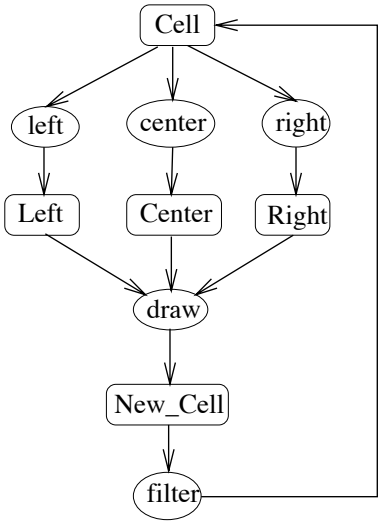


Figure 5.16: A network representing the cellular automata program given in the text.

This composition was straightforward. In the cellular automaton example of Section 5.2.5, we demonstrated that Γ can also capture the behaviour of cellular automata. This example is a first step towards what we hope will

be a thorough understanding of cellular automata in the context of multiset transformation. We leave further investigation of this topic for future work.

The Γ model is data-driven. That is, the function which will next be applied to the multiset is determined by the availability of data with the appropriate properties. Only when boundary conditions come into play or sequential composition is used is this freedom from communication lost. This data-driven behaviour of the model gives rise to a number of interesting effects. Firstly, in the case of our branching structures, it is not necessary that all branches of a given generation be processed by all functions before the first branch of the following generation is so passed. Rather, as long as a branch has been processed by a function, then it can be processed by the next. There is no need for each branch to be consumed as soon as it is generated, and no need for the branches to be consumed in the same order as that in which they were generated. We can therefore view Γ as a language for describing asynchronous process networks whose arcs are multisets, not streams, and whose selection of elements from those multisets is made non-deterministically. If we wish to force a more synchronous execution model, we can do so by including more sequential composition operators, or using the mechanism for introducing new multisets which contain flags to ensure the desired synchronisation. These issues are investigated in greater detail in Chapters 2 and 3.

Chapter 6

Conclusions

This thesis contains a mixture of theory and applications. The practical part examines ways of encoding, in multiset transformation languages (MSTLs), a number of physically- and biologically-motivated applications. The theory concentrates on the semantics of program composition operators (particularly interleaving and parallel operators) of MSTLs. We demonstrate that the details of interleaving or parallel operators in an MSTL greatly influence the possible behaviours of programs and hence influence the applicability of the language. The link between the application work and the theoretical work is this: the applications *drive* the theoretical work, suggesting ways in which current MSTLs are advantageous or deficient. The theoretical work then concentrates on ways of retaining the advantages of multiset transformation (MST) while overcoming any disadvantages highlighted by the applications. Along the way, we find that the interaction between these two approaches yields a number of interesting insights into what multiset languages are and what they should be. We also gain a better understanding of what parallelism and interleaving really are and the relationship between them. We claim that this mixed approach (theoretical computer science and computational science) yields a useful symbiosis: we obtain a collection of well-understood and formally-defined language constructs which we know can be used to build real applications, of interest to the physical and biological modelling communities. By using the applications to motivate the theoretical investigations, we therefore hope to avoid either researching the theoretical underpinnings of a poorly-applicable model, or of concentrating on building models using a formalism which may not be optimal, with all the frustrations thereby entailed.

Every chapter in this thesis contains its own conclusions, which will not be repeated here. Instead, we present a general summary of the work which has been described in this thesis. In particular, we draw the readers' attention to the connections between the work presented in the different chapters and indicate the ways in which the threads of the research can be drawn together into a tapestry. The tapestry is not complete, but the main scenes are clear, and enough has been woven to identify the main characters and their rôles in the story.

This chapter is in three parts: a summary of the other chapters in this thesis, indications of possible directions for future work and some concluding remarks.

6.1 Summary of the chapters

While the world of multiset transformation is relatively homogeneous when compared to, for example, the world of user interfaces, a number of variants of multiset transformation (MST) exist. Each has its own distinctive characteristics, although all share certain similarities. While this plethora of possibilities makes it possible to address many different issues in MST, the disparate notations used make it more difficult to compare the different languages and, therefore, to say something coherent about MST in general. In Chapter 2, we presented a unified model of parallel transformation and showed that this framework could be used both to provide an encoding for many MSTLs and to appreciate the differences between, and the difficulties associated with, their interleaving operators. We regard the most important contribution of Chapter 2 to be insight into the differences between alternative forms of program context-sensitive interleaving operators, together with the ability to generate a new operator by choosing the correct parameters for PT. These parameters, which indicate how 'interesting' different reductions are (and therefore how likely they are to occur), have both a formal and an informal component. The formal component is manifested in the semantics of PT, wherein more interesting reductions are chosen over less interesting reductions, when more than one reduction is possible. The informal component is the conceptual transparency of the notion of 'interesting' reductions which enables one easily to generate the parameters, and therefore a formal semantics, for a language whose program context-sensitive behaviour is understood intuitively. As a vehicle for exploring properties of different interleaving operators within an MSTL, we believe that PT has great value. To justify this conclusion, Chapter 2 contains, in terms of PT, encodings of a number of MSTLs, together with formal examination of the relationships between them.

In Chapter 3, we investigate, in terms of PT, ways of extending MSTLs so that a greater variety of applications can easily be encoded in them. Our motivation comes from encoding 'parallel' rewriting models such as L-systems, cellular automata, lattice-gas methods, N-body problems and many other physical applications. We argue that our encodings, which can be seen in Chapter 5, demonstrate that a number of extensions to MSTLs should be made if MSTLs are to be applicable to these problem areas. In particular, these applications require a more flexible approach to parallel rewriting with shared contexts than that available to conventional MSTLs such as Γ.

Most production systems (such as L-systems) do not make it possible to build 'real' programs; instead, a number of functions (or productions) are considered *implicitly* composed in parallel. One might even wish to argue that this is the whole *point* of production systems, viz. to leave implicit the model's operational details. Lacking explicit program composition opera-

tors, with their associated semantics, requires that implementations adhere to *ad hoc* 'standards' to decide which production to apply at each step. With MSTLs, on the other hand, programs contain explicit program constructors such as sequential or parallel composition. Each has its own formal semantics, which thereby disambiguates control-flow issues and opens them up to analysis.

However, widespread use of MSTLs in place of systems such as L-systems will only come about if MSTLs make it possible to express such constructs as logical simultaneity of reductions. Furthermore, most rewrites in systems like L-systems happen in the presence of data, which are examined but not rewritten. This is a *data context*, in the terminology of Chapter 3. An example of this is the examination of neighbours in a cellular automaton (see Chapter 3). As systems such as L-systems, lattice-gas etc. possess data context-sensitive rewriting, our MSTL has to do so too, in order to be able easily to encode such applications. Before the work described in this thesis was undertaken, writing MSTL programs involving either data contexts or explicit state update was very awkward for a programmer. For example, programming a cellular automaton in Γ requires writing a lot of code to ensure that every array element to be rewritten has the correct data context and that no element is rewritten more than once during a single generation of the automaton (see Chapter 5 for examples of this). In such cases, Turing completeness is a very poor argument for using a formalism. Rather, it should be easy to do the things which one often wants to do.

We encode the desired MSTL extensions in PT (our unifying formal framework from Chapter 2), by introducing new selection functions (selection functions are also explained in Chapter 2). These selection functions allow us to express programs which delay state updates until an explicit state synchronisation function is encountered. They also allow functions to make use of data contexts. These selection functions allow us to draw a distinction between parallel and interleaving operators, based upon their synchronisation behaviour in the presence of data contexts. We also argue that neither data contexts nor explicit synchronisation have received the attention they deserve in the MST community, perhaps because their usefulness has not be fully appreciated. We demonstrate that our new class of languages (at least one for every interest tuple: interest tuples are also explained in Chapter 2) is strictly more expressive than the class which makes use of Γ's selection function, by translating Γ into a language making use of our new selection functions and by giving an example in which our new formalism gives results which Γ cannot give. With these extensions of traditional MSTLs at our disposal, we are able to encode all of the problems which we mentioned at the beginning of the chapter (e.g. L-systems, cellular automata etc.).

Finally, another possible advantage of the new selection functions is mentioned: it may be that the possible parallelism available in a multiset program which makes use of data context might be higher than that of a program which does not make use of data context. If this turns out to be the case in general, it is a strong motivator for including data context in future multiset transformation languages.

At the end of this final theoretical chapter, we draw together the work described in both theoretical chapters by proposing a new MSTL, Goblin. Goblin delivers a single MSTL in which the developments of Chapters 2 and 3 are incorporated. Goblin's interleaving operator is completely context-insensitive and recursion is specified explicitly. These properties lead Goblin to avoid the compositionality difficulties associated with Γ [20] and CGP [37], wherein a programmer has to know which composed programs contain sequential composition, in order to understand what their behaviours will be. Furthermore, Goblin also makes available data context and explicit state synchronisations, which our work has shown to be crucial for readily implementing a number of models of biological and physical systems. Goblin, unfortunately, takes us somewhat away from the 'production system' spirit of Γ and CGP, in particular by making recursion explicit, but the prize is that composed programs are, arguably, better behaved than in either Γ or CGP. The behaviour of Goblin's interleaving composition, for example, is identical in all cases: choose a program and reduce it by one step.

Of course, Goblin programs can still interact in strange ways if they place data into or read data out of the same multiset(s), but this problem seems to this author to have more to do with the lack of a sensible module or object system than a basic semantic difficulty. If modules cannot be used to hide data from other programs, (and object-oriented programming has taught us that they can be so used), then interaction between context-insensitive programs seems unavoidable.

The theoretical work, then, gives us a new understanding of the relationships between those factors (data context, explicit synchronisation, interleaving, logical parallelism) which our practical investigations encourage us to examine. Goblin possesses a collection of operators which enable us to encode the programs which we want to encode. So far, the 'language' Goblin is little more than an annotated abstract syntax; development of this into a fully-fledged programming or modelling language is beyond the scope of this thesis.

The second part of the thesis consists of the two applications chapters. Chapter 5 contains the examples which we first encoded into Γ. These examples motivated the theoretical investigations of the first part of the thesis. Chapter 4 gives an analysis, in terms of Goblin, of ways of encoding stochastic applications into MSTLs. The examples given in both chapters originated from a desire to enlarge the set of examples available to the MST community. Along the way, they highlighted shortcomings over existent MSTLs, particularly with regard to problems of recursion, data contexts and synchronised state updates. We have already discussed these shortcomings and the work which was undertaken to overcome them. The belief that these shortcomings were fundamental and not cosmetic inspired the theoretical work described in the earlier chapters, which solved the problems (with the exception of data structuring).

Chapter 4 is an analysis of ways of harnessing probabilities to make it possible to code stochastic applications in MSTLs, without requiring that any additions to the language be made. This work contradicts those who

claim, or imply, that stochastic non-determinism is available from MSTLs only when an extension to the model is made (see, for example, [108]). Particular areas which were examined include influencing the choice of elements from the multiset, the choice of functions in an interleaving composition and control of the program's termination. All of these choices can be made stochastic, a claim which we highlighted with a number of short examples. We also gave examples of Monte-Carlo integral approximation, diffusion-limited aggregation clusters, genetic algorithms and simulated annealing, to substantiate our claims.

Chapter 5 gives examples of environmentally-sensitive growth forms, seeded and grown within a multiset. Our examples include photosensitive growth forms, self-avoiding structures and combinations thereof. Most of these forms are similar to those generated using L-systems, although our examples feature branches grown without regard to the 'simultaneous' growth of branches in real organisms. In the examples given, it is possible for biologically impossible situations to occur: an older branch might never develop because it is impinged upon by a younger branch. This limitation motivated the search for a solution to the problem, which is solved in Chapter 3.

The other main example of this, last, applications chapter is a cellular automaton. Like an L-system, a cellular automaton features logically simultaneous, data context-sensitive rewrites of all cells of a particular generation. In the current example, which is written in Γ, it is necessary that the programmer go through all sorts of contortions to explicitly generate the context for each cell and then to ensure that all cells of one generation are rewritten before any of the succeeding generation. This is ridiculous. With the extensions suggested in the theoretical half of the thesis, the situation has been greatly improved. Furthermore, with the exception of data structures in the multiset, all other problems which were discovered in Γ as a result of writing these example programs, were solved in Chapters 2 and 3.

6.2 Future work

Despite the general tone of optimism in many of the above remarks, every cloud has a silver nitrate[1]. In particular, much work still remains to be done. We mention here some of the unresolved issues.

6.2.1 Theoretical tasks ahead

The theoretical tasks ahead have become clearer, now that the issues of recursion, program and data contexts and synchronous and non-synchronous state update, have been addressed. From here, some language-design work could take us from Goblin, which is basically an annotated abstract syntax, into a real coordination *language*. We have not attempted to do this in this thesis.

A number of theoretical issues remain in need of clarification. Firstly, how do we formally describe non-atomic primitive functions? Functions (i.e.

[1] Like a silver lining, but much worse.

software components) which take a long time to produce output must be able to be executed logically in parallel with other functions. So far, our semantics assumes that primitive functions execute quickly enough that the system can wait until they have produced output (or aborted or deadlocked) before deciding what to do. Our discussion of logical parallelism in Chapter 3 still makes this (unrealistic) assumption. We hope that a formal treatment of this issue would address the requirement that programs 'lock' part of the non-contextual state while executing, without requiring other programs to wait for the laggard.

6.2.2 Practical tasks ahead

In open systems [41], functionality can be removed from or added to the system at runtime. We are not yet able to model this in PT. Our programs so far bear little relationship to the software objects which we expect as components of modern systems. Given the almost universal use of object-orientation and openness in industrial systems, the current MST orientation seems somewhat quaint.

It is not the intention here to give the impression that such a dynamic model as open systems cannot *a priori* be described in MST (actually, it might be relatively straightforward). Rather, such a model implies a change in the way that we think about the nature of our programs and what it is, precisely, that we are coordinating. These issues have not been addressed in this thesis, for the simple reason that the motivation here leans more towards physical and biological modelling rather than towards software methodologies and engineering.

Whatever the philosophical orientation taken, more serious implementations and applications of MSTLs are needed. In particular, large numerical, symbolic and industrial projects must be undertaken, for two reasons.

1. To convince the wider community that MSTLs are serious alternatives to be considered in industrial-strength problem solving.

2. To convince the MST community that the MSTL design decisions which are being taken are the correct ones, in the sense of being convenient for the user, implementable and applicable. As this thesis has demonstrated, the results of designing a model to fit the applications are very different to (and sometimes at odds with) the results of designing models without such an application-driven perspective.

It is important not to underestimate the importance of providing (or attempting) to provide, serious implementations and applications of MSTLs. As the development of functional languages has shown us, initally optimistic hopes for easy and, perhaps, parallel implementations of functional languages were shown to be unfounded. Only in the last few years have languages such as Haskell [135] and Sisal [54] shown us that efficiency in functional language implementations, can be realised. That it has been a harder task than many first hoped, should cause us to be wary of claiming too much for MST with

too little implementation experience. In the course of the work described in this thesis, a number of prototype implementations of various MSTLs were built in order to be able to run the example programs (the graphics seen throughout this thesis are genuine, being generated by any of a number of Γ and Goblin implementations). Building a real compiler is an interesting challenge for the future. Issues such as scalability of implementations (and of programs) cannot be sensibly discussed in the absence of a good, optimising, compiler.

6.2.3 The multiset: there is no silver billet[2]

Throughout this thesis, we have said very little about how to structure the data in a multiset, although we mentioned the problem in the introduction. Currently, the multiset is a space containing only tuples, so that the relationships between, for example, cells in a cellular automaton, have to be explicitly encoded. This is clearly unsatisfactory: the multiset, rather than helping us to express the problem, hinders us by not allowing us to express the structure inherent in the data. Structuring the multiset to increase the programmability of MSTLs has become a bit of a *cause celébre* in recent years, with a number of researchers addressing the issue [97, 53]. Unfortunately, the problem is not yet solved to the satisfaction of all concerned, which limits the use of MSTLs to areas where data structure is minimal or the programmer is particularly motivated. It seems to this author that the problems of data structuring in MSTLs can be solved if we regard the multiset not as the mother of all data structures but as an augmentation of the traditional repertoire of data structures. Such an approach would mean that we could use arrays, references and so on, in addition to multisets. Multisets could then be used as 'don't care' data structures while other, more traditional, data structures could be used when data relationships are important. A combination of multisets and traditional data structures would enable one to build random access data structures, where all elements reside in a multiset (and are therefore simultaneously visible to the program) but are interconnected using traditional techniques (e.g. references), so that accessing one element allows one quick and easy access to all of that element's 'neighbours' (for example). Although not examined in this thesis, these issues should be investigated.

6.3 Exeunt

We have formalised a unified framework for MST called PT, into which we have translated a number of existing MSTLs and proposed some new MSTLs. The relationships between these MSTLs has been investigated, in terms of PT. The investigations have lead to a better understanding of a variety of program context-sensitive and insensitive interleaving operators. We have examined a number of applications of MSTLs within the physical and biological modelling fields. On the basis of these applications we have investigated

[2] With apologies to Ken Brookes.

extensions to current MSTLs which enable us to capture the properties that we require in order to be able readily to encode such applications. These properties include logically simultaneous state updates (which we hope an implementation can turn into actual parallelism) and data context-sensitive reductions. We have proposed a new MSTL, Goblin, which contains all the properties that we claimed we should have. We have investigated methods for encoding different stochastic applications within Goblin, illustrating our claims with a number of examples from the physical modelling world. We therefore have a formally-defined, applicable basis for programming, using only a multiset as a data structure.

This thesis has explored some of the issues which the author believes must be addressed if MST is to be a powerful and flexible tool for coordinating software components. It is hoped that the formal nature of the investigations, together with their motivation by, and application to, problems in physical and biological modelling, are a good foundation for yet more useful models. A number of issues remain to be resolved (see Section 6.2), but we have made useful progress.

Multiset transformation has come a long way in the last ten years. It has developed from a proposal outlined in a handful of papers and has become an industry (albeit a cottage industry) and a major player in conferences discussing coordination and parallel programming. If there is one thing which writing a thesis leads this author to believe, it is this: in order to produce a coordination model which is of lasting use to the real world, we have to make sure that the formalisms we produce can be used to write serious, industrial applications. It is not enough to define a language and then to claim 'This is it. Can we use it?'. A language has to be designed with its applications in mind. Such an approach might help ensure that the language can be used for something other than for generating publications. We need MSTLs which can be used as real programming or modelling languages, with industrial-strength implementations, debuggers, parallelising compilers for local-area networks (LANS) and distributed-memory, parallel, machines. To build and test these MSTLs of tomorrow and of the day after tomorrow, we need a collection of well-understood formal transformations and analyses. We also need suites of large, tested, bench-marked programs. In this author's opinion, it is only by following such a route that industry and academia can ensure that MSTLs become technology that is really used by people and is not just another corpse, left quietly to decay, in the gutter by The Road Ahead.

I have a dream.

Chapter 7

Samenvattingen/summaries

7.1 Nederlands

Dit proefschrift gaat over de resultaten van mijn promotieonderzoek in de semantiek en toepassingen van een klasse van computationele modellen, die 'multiset transformation languages' (MSTL's) heet. Een MSTL bestaat uit een globale staat (de meervoudverzameling) en een aantal functies die transformaties van meervoudverzameling tot meervoudverzameling leveren.

Het blijkt dat MST-achtige concepten in veel takken te vinden zijn. Omdat deze concepten in zoveel disciplines voorkomen, blijkt het dat MST waardevol is. Om dit kracht bij te zetten, wordt in Hoofdstuk 1 een overzicht van de verschillende variaties en varianten van MST gegeven. In dit hoofdstuk wordt ook een discussie over de relaties tussen verschillende MSTL's van een aantal verschillende onderdelen in de informatica gegeven. Het verhaal probeert de gezamelijke eigenschappen van al de verschillende MST-achtige modellen en talen helder te maken, en ook een duidelijke presentatie van hun verschillende eigenschappen te geven. Daardoor wordt b.v. een vergelijking uitgevoerd tussen het werk van David Sands (over semantiek) en werk dat door Walter Fontana over de creatie van het leven gedaan is. Zo breed is het wereld van MST: het komt overal voor.

Mijn ondezoek is een poging om MST's nog meer waarde te geven. Bestaande MST's kunnen onderverdeeld worden in hoofdzakelijk twee groepen. De eerste is een groep talen die formeel-gedefinieerd zijn (b.v. Γ, CGP), maar die geen goede compilers of toepassingen hebben. De andere groep is meer 'pragmatisch' gebouwd (b.v. Linda). Deze klasse heeft geen envoudige formele semantiek, maar wel implementaties en/of applicaties. De laatste vier jaar, was een poging gedaan om een 'unificatie' tussen die twee klassen te maken, waarmee zowel formele als gecompileerde/toepassing gerichte talen kunnen worden gebouwd. Uiteindelijk wordt er een 'speeltuin taal' gemaakt, die de goede eigenschappen van beide werelden heeft. Voorbeelden van programma's zijn ook gegeven, die in deze taal geschreven kunnen worden.

Dit proefschift valt uiteen in twee delen. Het eerste deel bestaat uit theoretisch onderzoek met betrekking tot de semantiek van interleaving constructies in MSTL's. Een interleaving constructie is een soort scheduler die, op

een bepaald moment, kiest welke van de programma componenten worden geëxecuteerd. Het gekozen programma (onderdeel) loopt een tijdje, waarna de interleaving operator wederom kiest of een lopend programma gaat slapen of dat een slapend programma wordt gewekt en gaat lopen. Dit werk wordt in twee hoofdstukken behandeld. Het eerste hoofdstuk concentreert zich op de classificatie en analyse van een aantal verschillende MSTL's, terwijl het andere hoofdstuk over een uitgebreid soort model gaat, waarin z.g. *data context* te gebruiken is. Data context wordt later uitgelegd.

Als een programma uit een interleaving compositie van meerdere sub-programma's is gebouwd, kan de volgende vraag gesteld worden: hoe kiest man welke programma('s) in de volgende tijdstap geëxecuteerd kunnen worden? Het blijkt dat de keuze die gemaakt wordt veel invloed op de mogelijke executies van de hele programma kan hebben. Bij voorbeeld, het programma dat de vorige keer gekozen was, kan altijd gekozen worden. Als alternatief, één van de programma's gekeusen kan worden die uit een sequentiele compositie van sub-programma's gebouwd is. In hoofdstuk 2, wordt een formeel framework gegeven waarin dit soort vragen geanalyserd kunnen worden. Dit framework is in termen van een z. g. Structured Operational Semantics (SOS) gedefinieerd, waarin parameters ingevuld kunnen worden. Verschillende parameters geven verschillende interleaving mogelijkheden. Het basis-idee is om te bepalen was de relatie is tussen een bepaalde soort interleaving operator en die eigenschappen van een taal die so een interleaving operator heeft. Interleaving operatoren die verschillend keusen maken op basis van de gecomposeerd programma's zijn 'program context sensitief'. Het hoofstuk bestaat uit een aantal voorbeelden van talen die in termen van de framework beschrijven kunnen worden en zodoende de verschillende mogelijkheiden kunnen vergelijken. Er wordt getoond, met behulp van meervoudige voorbeelen, dat program context sensitiviteit in het algemeen geen goede eigenschap van een interleaving operator is.

Hoofdstuk 3 bestaat uit een studie van de invloed van data context op de implementatie en executie van programma's. Data context is een soort 'beeld' van de data dat niet van de meervoudverzameling afkomstig is en dus voor alle sub-programma's zichtbaar is. Af en toe moet een nieuw 'beeld' van het geheugen genomen worden. Dit nemen van een 'snapshot' heet een 'synchronisatie'. Met data context is het veel makkelijker om modellen van ge-synchroniseerd fysische processen te modelleren, b.v. de groei van planten en bloemen. Men kan onze uitbreiding van interleaving met data context als een vorm van parallellisme zien. Om te bewijzen dat dit nieuwe framework (met data context) formeel meer algemeen is in vergelijking met een framework zonder data context, wordt er een voorbeeld en een formeel bewijs gegeven. Het toont aan hoe sommige mogelijke antwoorden onmogelijk worden, indien geen data context beschikbaar is. Met het bewijs wordt er getoond hoe een 'gewone' interleaving in het model toegepast kan worden (namelijk: na iedere funktie applicatie wordt er een synchronisatie gedaan).

Het einde van Hoofstuk 3 presenteert een nieuwe MSTL: Goblin. Goblin heeft al de goede eigenschappen die in de laatste twee hoofdstukken bepaald zijn geweest, tevens mist het de besproken slechte eigenschappen. Dus heeft

Goblin een niet-context sensitive interleaving operator, data context en expliciete synchonisatie functies.

De rest van het proefschrift bestaat uit de meer praktische kanten van het werk: een onderzoek naar hoe het mogelijk zou zijn om probabilistische toepassingen in een MSTL te schrijven en een 'teken-boek' van voorbeelden die uit de modelleringswereld komen.

Hoofdstuk 4 geeft een antwoord op de vraag 'hoe kan je stochastische programma's in een MSTL schrijven?'. Het antwoord bestaat uit twee delen. Ten eerste worden alle verschillende manieren beschreven om een stochastisch programma in een MSTL te schrijven. Dit laat zien dat het onnodig is om probabilistische primitiven *in* een MSTL te stoppen, omdat in principe alles expliciet in die programma's geschreven kunnen worden. De tweede helft van het antwoord beschijft een aantal meer serieuze voorbeelden uit de modelleringswereld. Dus is er niet alleen de mogelijkheid van encodering, maar ook van de actualiteit.

Hoofstuk 5 is het laatste technische hoofdstuk. Het geeft voorbeelden van modellen van fractale groei processen, lopende van eenvoudig (een boom) tot meer complex (een licht-sensitieve boom waarvan de takken niet tegen elkaar kunnen groeien). De complexe voorbeelden corresponderen natuurlijk meer met de natuurlijke wereld. De eenvoudige voorbeelden geven een basis waarmee de minder eenvoudige voorbeelden gebouwd kunnen worden. Dit hoofdstuk is het laatste 'echte' hoofstuk, maar het gaf eigenlijk de inspiratie voor het hele onderzoek. Met de besproken voorbeelden, is het de mening van de auteur dat MST en MSTL's waardevol zijn indien hun nadelen maar overkomen kunnen worden.

Uiteindelijk komt het laatste hoofdstuk, Hoofdstuk 6. Deze bestaat uit een aantal conclusies, open problemen en toekomstig werk. Een kort overzicht van het hele proefschrift wordt gegeven, samen met wat meer subjectief commentaar over het hele gevolgde traject.

En Bob is je oom, zoals die Engelse zeggen!

7.2 English

This summary is for all those friends and family members who have asked or wondered 'just what is Hugh doing in Amsterdam?'[1]. Computer science (like all disciplines) is snowed under with terminology, notations and a great deal of other 'trade baggage' which makes it extraordinarily difficult for a lay-person to understand what all of these people are talking about. An academic thesis is particularly prone to this difficulty being, as it is, an in-depth, technical study of a very specialised area, written by someone who is probably not yet experienced enough to be able to present the most difficult material in a clear way. The combination of these two properties frequently leads the non-specialist reader astray. This section constitutes an attempt to explain what I have been up to here in a way which, I hope, those dearest to me can understand.

[1] This is a question I have asked myself many times.

This thesis concerns itself with investigations of a class of computer languages called multiset transformation (MST) languages (MSTLs). The investigations fall, broadly, into two main areas. Firstly, textually speaking, come the theoretical investigations of MSTLs in Chapters 2 and 3. These chapters are mathematically the most dense of the thesis, concerning themselves, as they do, with very abstract models of what constitutes a certain kind of computation. The second 'half' of the thesis consists of investigations of applications for MSTLs, about which I shall say more in due course.

The *theory* of MSTLs is a mathematically-precise description of the way in which computer programs, written in these languages, will execute. That is, for an MST program applied to a particular state (a *state* is just the contents of part of the computer's memory at a particular moment), the mathematical description will tell us precisely the state (or states: there may be more than one possible) which can follow from the current one. From each new possible state, we can calculate what the next state(s) from that can be. In this way, given the initial state of the program, we can calculate all possible sequences of states which the program can reach. We can therefore say that, in some sense, the *meaning* of the program is given by the way in which it can pass from state to state. In this thesis, the mathematical descriptions are given in a form known as a Structured Operational Semantics (SOS).

Chapters 2 and 3 concern themselves with theoretical issues. In Chapter 2, we give an SOS into which parameters can be inserted. Depending upon the choice of parameters inserted, it is possible for the semantics(the SOS) to describe a great variety of different languages. This chapter mainly concerns itself with *interleaved* programs. If two program executions are interleaved then (informally), at every moment one of them will be executing. They will not both be executing at the same time, but each can stop and let the other take over from time to time. This is essentially the same as what one does when trying to do two jobs 'at the same time': do a bit of the first, then some of the second, then more of the first etc. Many computer programs execute this way because most computers have a single microprocessor but are trying to do several things at once (e.g. draw a picture on the screen while following the operator's typing while printing a document, etc.). All this prompts the question: how do we choose which task to work on at a given moment? In Chapter 2, we show how our theoretical framework (our parameterisable SOS) can be used to capture a number of different possible strategies for choosing which program to execute next. Our framework enables us to compare different approaches in a formal way. The investigations yield some insight both into new ways in which programs can be executed and into how the old ways are related to each other.

The second theoretical chapter concerns another class of program, where (i) programs have to look at some part of the state which they are not going to change (we call this *data context*) and (ii) several programs wish to execute together, rather than one at a time. This latter property is called *parallelism*. For many important application areas for computers, it is much easier to write our programs if we have these two properties at our disposal

than if we do not. We give examples of such programs throughout Chapters 3 and 5. MSTLs have not, until this work, possessed both of these properties, so it is interesting to see the useful things that they can do once they are extended so as to have them.

The difference between interleaving and parallelism is subtle but fundamental, when data context is present. We illustrate the distinction. Imagine that two cars are driving in opposite directions along a road and are travelling towards one another. In front of each car is a mad cow, which is moving erratically and very slowly along the road. Both cars wish to overtake the cows. What could happen when the cars try to overtake clearly illustrates why interleaving and parallelism are not always the same. Imagine that both drivers overtake using the following rule (often used by lazy drivers) 'if the road is clear, overtake immediately'. Imagine also that the drivers of both cars perform their observations of the state of the road at the same time. Both drivers will then start to overtake immediately, with the result that an accident is possible. So, even though both drivers 'look before they leap', what they have seen represents the state of the world in the past. They are observing and reacting simultaneously and therefore in parallel. Now, consider an alternative scenario in that one of the drivers has to perform her observation before the other one, but the observations can happen in any order. The driver who observes first will then immediately begin to overtake. The other, who performs her observation later, will then see that the other driver is already overtaking and will not attempt to overtake. No accident can result (assuming (i) that the first driver reacts *immediately* after observing the road and (ii) this action is immediately visible to the other driver). For those of you who have driven in the USA, notice that this corresponds exactly to the problems raised at a crossroads, where the car arriving first has priority. If several cars arrive in any order, no problems will result (this is interleaving). However, if two cars arrive at *exactly* the same moment, everyone will give way to everyone else and traffic will grind to a halt (this is parallelism, but not very productive parallelism).

At the end of Chapter 3, a new MSTL is presented. Christened Goblin (because it's small, ugly and has teeth), the language attempts to avoid the problems of contemporary MSTLs, which were described in Chapters 2 and 3. The result is a language possessing (arguably) a well-behaved interleaving operator and awareness of data context. The former avoids certain difficulties suffered from by languages such as Γ and CGP. The latter (awareness of data context) makes it possible for the programmer to clearly see exactly where parallelism and interleaving can be used in an implementation and enables her to specify which of these she wants to use and when.

Chapter 3 represents the last of the theoretical chapters (whew!). The rest of the thesis (with the exception of the conclusions) concerns itself with more practical issues. Chapter 4 asks the question 'Can we encode stochastic applications in MST?'. The answer is 'Yes'; we give examples showing a number of ways in which someone can write such programs in an MSTL. We isolate six different approaches one might wish to take, giving examples of

how to encode each. At the end of the chapter, we give a few larger examples to show how our ideas can be applied to 'real' examples. The examples are still small, but represent examples of problems that are of interest to the physical modelling community.

Chapter 5 is a picture-book of examples of biological models built using MST. Various programs are given, starting with the simplest model (a fractal tree) and ending with a model of photosensitive (light-sensitive) growth of an organism whose branches cannot cross each other. The latter example corresponds more closely to growth in the natural world, while the former merely illustrates the principles underlying the more complex examples. At the end of the chapter, we give an example of a cellular automaton; a particular form of environmentally-sensitive growth which takes place on a lattice (a grid). Cellular automata are widely used in the physical modelling community to investigate complex emergent effects associated with interactions between simple entities. Examples include the macroscopic effects caused by gas molecules colliding with each other (e.g. sound transmission) and models of animal population growth and decay (the most famous example of which is Conway's Game of Life).

The thesis concludes with Chapter 6, which sums-up the work which has been done, the results obtained and the possible avenues for future research. We end on a philosophical note, with a dream of a better tomorrow.

And that is the whole egg-eating, as the Dutch say.

Appendix A

Proofs

This appendix contains three inductive proofs. The first two prove propositions stated in Chapter 2, while the third proves a proposition stated in Chapter 3.

All the proofs are written in Lamport's structured proof style, introduced in [89]. This proof style makes use of a natural-deduction [99] style of proof, augmented with a numbering scheme which emphasises the proof's structure. The level of indentation is indicated by a number within angle brackets, thus: $\langle$ x $\rangle$. Numbers after the angle brackets indicate the current line number. Discharging of assumption number n at level x in the proof is written $\langle$ x $\rangle$: n. References to other lines, for example when justifying a deduction step, always refer to the *nearest* line with that number before the current line.

A.1 Γ to PT proof

We repeat and prove Proposition 1.

Proposition 10 (Correctness of the translation) *Our translation of* Γ *into PT is correct. Formally, and writing* $\longrightarrow_\Gamma$ *as the* Γ *transition relation and* $\Longrightarrow$ *as the PT transition relation, we have:*

$$\begin{array}{lll} \forall P, Q, M, N. & \langle P, M\rangle \longrightarrow_\Gamma \langle Q, N\rangle & \textit{iff} \quad \langle [\![P]\!], M\rangle \Longrightarrow^+ \langle [\![Q]\!], N\rangle \wedge \\ & \langle P, Q\rangle \longrightarrow_\Gamma M & \textit{iff} \quad \langle [\![P]\!], M\rangle \Longrightarrow^+ \langle \epsilon, M\rangle \end{array}$$

The proposition is formalised below and the proof given by induction over the structure of the transition relation.

Our proof is in two parts. The first part is a proof that the proposition holds for all *simple* Γ programs. Simple Γ programs contain no sequential composition [65]. The second part extends the result to all non-simple programs.

A.1.1 Simple programs

Base case: primitive functions.

Successful primitive function application.

$\langle 1\rangle 1$. Prove $\langle (B, A), M\rangle \longrightarrow_\Gamma \langle (B, A), N\rangle$ iff
$\langle (B, A, \mathcal{S}_\Gamma)^*, M\rangle \overset{(\mathtt{succ},\mathtt{succ})}{\Longrightarrow} (\mathbf{0},\infty)^+ \langle (B, A, \mathcal{S}_\Gamma)^*, N\rangle$
$\langle 2\rangle 1$. Assume $\langle (B, A), M\rangle \longrightarrow_\Gamma \langle (B, A), N\rangle$.

Prove $\langle (B,A,\mathcal{S}_\Gamma)^*, M\rangle \overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle (B,A,\mathcal{S}_\Gamma)^*, N\rangle$.

$\langle 3\rangle 1$. $\exists \vec{m} \subseteq M.B(\vec{m}) \wedge N = M[A(\vec{m})/\vec{m}]$, Assumption $\langle 2\rangle$ and Γ SOS.

$\langle 3\rangle 2$. $\langle (B,A,\mathcal{S}_\Gamma)^*, M\rangle \overset{(\mathtt{succ},\mathtt{succ})}{\Longrightarrow} \langle \epsilon \overset{\mathtt{succ}}{\triangleright} (B,A,\mathcal{S}_\Gamma)^*, N\rangle$, from $\langle 3\rangle 1$ and PT SOS.

$\langle 3\rangle 3$. $\langle \epsilon \overset{\mathtt{succ}}{\triangleright} (B,A,\mathcal{S}_\Gamma)^*, N\rangle \overset{(\mathbf{0},\infty)^+}{\Longrightarrow} \langle (B,A,\mathcal{S}_\Gamma)^*, N\rangle$, from PT SOS and $\mathtt{if} < \mathtt{succ}$.

$\langle 3\rangle 4$. $\langle (B,A,\mathcal{S}_\Gamma)^*, M\rangle \overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle (B,A,\mathcal{S}_\Gamma)^*, N\rangle$, from $\langle 3\rangle 2$ and $\langle 3\rangle 3$ and transitivity.

$\langle 3\rangle 5$. Q. E. D.

$\langle 2\rangle 2$. Assume $\langle (B,A,\mathcal{S}_\Gamma)^*, M\rangle \overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle (B,A,\mathcal{S}_\Gamma)^*, N\rangle$.
Prove $\langle (B,A), M\rangle \longrightarrow_\Gamma \langle (B,A), N\rangle$.

$\langle 3\rangle 1$. $\exists \vec{m} \subseteq M.B(\vec{m}) \wedge N = M[A(\vec{m})/\vec{m}]$, Assumption $\langle 2\rangle$ and PT SOS.

$\langle 3\rangle 2$. $\langle (B,A), M\rangle \longrightarrow_\Gamma \langle (B,A), N\rangle$, from $\langle 3\rangle 1$ and Γ SOS.

$\langle 3\rangle 3$. Q. E. D.

Unsuccessful primitive function application.

$\langle 1\rangle 2$. Prove $\langle (B,A), M\rangle \longrightarrow_\Gamma M$ iff $\langle (B,A,\mathcal{S}_\Gamma)^*, M\rangle \overset{(\mathtt{fail},\mathtt{fail})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, M\rangle$

$\langle 2\rangle 1$. Assume $\langle (B,A), M\rangle \longrightarrow_\Gamma M$.
Prove $\langle (B,A,\mathcal{S}_\Gamma)^*, M\rangle \overset{(\mathtt{fail},\mathtt{fail})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, M\rangle$.

$\langle 3\rangle 1$. $\neg\exists \vec{m} \subseteq M.B(\vec{m})$, Assumption $\langle 2\rangle$ and Γ SOS.

$\langle 3\rangle 2$. $\langle (B,A,\mathcal{S}_\Gamma)^*, M\rangle \overset{(\mathtt{fail},\mathtt{fail})}{\Longrightarrow} \langle \epsilon \overset{\mathtt{fail}}{\triangleright} (B,A,\mathcal{S}_\Gamma)^*, M\rangle$, from $\langle 3\rangle 1$ and PT SOS.

$\langle 3\rangle 3$. $\langle \epsilon \overset{\mathtt{fail}}{\triangleright} (B,A,\mathcal{S}_\Gamma)^*, M\rangle \overset{(\mathbf{0},\infty)}{\Longrightarrow} \langle \epsilon, M\rangle$, from $\mathtt{if} > \mathtt{fail}$ and PT SOS.

$\langle 3\rangle 4$. $\langle (B,A,\mathcal{S}_\Gamma)^*, M\rangle \overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle (B,A,\mathcal{S}_\Gamma)^*, N\rangle$, from $\langle 3\rangle 2$ and $\langle 3\rangle 3$ and transitivity.

$\langle 3\rangle 5$. Q. E. D.

$\langle 2\rangle 2$. Assume $\langle (B,A,\mathcal{S}_\Gamma)^*, M\rangle \overset{(\mathtt{fail},\mathtt{fail})(\mathbf{0},\infty)}{\Longrightarrow} \langle \epsilon, M\rangle$.
Prove $\langle (B,A), M\rangle \longrightarrow_\Gamma \langle (B,A), M\rangle$.

$\langle 3\rangle 1$. $\neg\exists \vec{m} \subseteq M.B(\vec{m})$, Assumption $\langle 2\rangle$ and PT SOS.

$\langle 3\rangle 2$. $\langle (B,A), M\rangle \longrightarrow_\Gamma M$, from $\langle 3\rangle 1$ and Γ SOS.

$\langle 3\rangle 3$. Q. E. D.

Inductive case: interleaved compositions of simple programs. If a primitive function in an interleaved composition of simple programs is applicable, then it will be applied in preference to an inapplicable primitive function.

Successful simple interleaving.

$\langle 1\rangle 3$. Prove $\langle P+Q, M\rangle \longrightarrow_\Gamma \langle P+Q, N\rangle$ iff
$\langle [\![P]\!] \,|||\, [\![Q]\!], M\rangle \overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P]\!] \,|||\, [\![Q]\!], N\rangle$.

$\langle 2\rangle 1$. Assume 1. $\langle P, M\rangle \longrightarrow_\Gamma \langle P, N\rangle$ iff $\langle [\![P]\!], M\rangle \overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P]\!], N\rangle$.
2. $\langle Q, M\rangle \longrightarrow_\Gamma \langle Q, N\rangle$ iff $\langle [\![Q]\!], M\rangle \overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![Q]\!], N\rangle$.
3. $\langle P+Q, M\rangle \longrightarrow_\Gamma \langle P+Q, N\rangle$.
Prove $\langle [\![P]\!] \,|||\, [\![Q]\!], M\rangle \overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P]\!] \,|||\, [\![Q]\!], N\rangle$.

$\langle 3\rangle 1$. $\langle P, M\rangle \longrightarrow_\Gamma \langle P, N\rangle$ or
$\langle Q, M\rangle \longrightarrow_\Gamma \langle Q, N\rangle$, Assumption $\langle 2\rangle$:3 and Γ SOS.

$\langle 3\rangle 2$. WLOG, assume $\langle P, M\rangle \longrightarrow_\Gamma \langle P, N\rangle$.

$\langle 3\rangle 3$. $\langle [\![P]\!], M\rangle \overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P]\!], N\rangle$, Assumption $\langle 2\rangle$:1 and $\langle 3\rangle 2$.

$\langle 3\rangle 4$. $\langle [\![P]\!] \,|||\, [\![Q]\!], M\rangle \overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P]\!] \,|||\, [\![Q]\!]), N\rangle$, from $\langle 3\rangle 3$ and PT SOS.

$\langle 3\rangle 5$. Q. E. D.

$\langle 2\rangle 2$. Assume 1. $\langle P, M\rangle \longrightarrow_\Gamma \langle P, N\rangle$ iff $\langle [\![P]\!], M\rangle \overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P]\!], N\rangle$
2. $\langle Q, M\rangle \longrightarrow_\Gamma \langle Q, N\rangle$ iff $\langle [\![Q]\!], M\rangle \overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![Q]\!], N\rangle$
3. $\langle [\![P]\!] \,|||\, [\![Q]\!], M\rangle \overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P]\!] \,|||\, [\![Q]\!], N\rangle$.

Prove $\langle P+Q, M\rangle \longrightarrow_\Gamma \langle P+Q, N\rangle$.
$\langle 3\rangle$1. $\langle [\![P]\!], M\rangle \overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P]\!], N\rangle$ or
$\langle [\![Q]\!], M\rangle \overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![Q]\!], N\rangle$, Assumption $\langle 2\rangle$:3 and PT SOS.
$\langle 3\rangle$2. WLOG, assume $\langle [\![P]\!], M\rangle \overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P]\!], N\rangle$.
$\langle 3\rangle$3. $\langle P, M\rangle \longrightarrow_\Gamma \langle P, N\rangle$, Assumption $\langle 2\rangle$:1 and $\langle 3\rangle$2
$\langle 3\rangle$4. $\langle P+Q, M\rangle \longrightarrow_\Gamma \langle P+Q, N\rangle$, from $\langle 3\rangle$3 and Γ SOS.
$\langle 3\rangle$5. Q. E. D.

Unsuccessful simple interleaving.

$\langle 1\rangle$4. Prove $\langle P+Q, M\rangle \longrightarrow_\Gamma M$ iff $\langle [\![P]\!] \,|||\, [\![Q]\!], M\rangle \overset{(\mathtt{fail},\mathtt{fail})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, M\rangle$
$\langle 2\rangle$1. Assume 1. $\langle P, M\rangle \longrightarrow_\Gamma M$ iff $\langle [\![P]\!], M\rangle \overset{(\mathtt{fail},\mathtt{fail})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, M\rangle$.
2. $\langle Q, M\rangle \longrightarrow_\Gamma M$ iff $\langle [\![Q]\!], M\rangle \overset{(\mathtt{fail},\mathtt{fail})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, M\rangle$.
3. $\langle P+Q, M\rangle \longrightarrow_\Gamma M$.
Prove $\langle [\![P]\!] \,|||\, [\![Q]\!], M\rangle \overset{(\mathtt{fail},\mathtt{fail})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, M\rangle$.
$\langle 3\rangle$1. $\langle P, M\rangle \longrightarrow_\Gamma M$, Assumption $\langle 2\rangle$:3 and Γ SOS.
$\langle 3\rangle$2. $\langle [\![P]\!], M\rangle \overset{(\mathtt{fail},\mathtt{fail})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, M\rangle$, Assumption $\langle 2\rangle$:1 and $\langle 3\rangle$1.
$\langle 3\rangle$3. $\langle Q, M\rangle \longrightarrow_\Gamma M$, Assumption $\langle 2\rangle$:3 and Γ SOS.
$\langle 3\rangle$4. $\langle [\![Q]\!], M\rangle \overset{(\mathtt{fail},\mathtt{fail})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, M\rangle$, Assumption $\langle 2\rangle$:2 and $\langle 3\rangle$3.
$\langle 3\rangle$5. $\langle [\![P]\!] \,|||\, [\![Q]\!], M\rangle \overset{(\mathtt{fail},\mathtt{fail})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, M\rangle$,
from $\langle 3\rangle$2 and $\langle 3\rangle$4 and $\mathtt{fail} < \mathtt{sync}$ and PT SOS.
$\langle 3\rangle$6. Q. E. D.
$\langle 2\rangle$2. Assume 1. $\langle P, M\rangle \longrightarrow_\Gamma M$ iff $\langle [\![P]\!], M\rangle \overset{(\mathtt{fail},\mathtt{fail})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, M\rangle$.
2. $\langle Q, M\rangle \longrightarrow_\Gamma M$ iff $\langle [\![Q]\!], M\rangle \overset{(\mathtt{fail},\mathtt{fail})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, M\rangle$.
3. $\langle [\![P]\!] \,|||\, [\![Q]\!], M\rangle \overset{(\mathtt{fail},\mathtt{fail})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, M\rangle$.
Prove $\langle P+Q, M\rangle \longrightarrow_\Gamma M$.
$\langle 3\rangle$1. $\langle [\![P]\!], M\rangle \overset{(\mathtt{fail},\mathtt{fail})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, M\rangle$, Assumption $\langle 2\rangle$:3 and PT SOS.
$\langle 3\rangle$2. $\langle P, M\rangle \longrightarrow_\Gamma M$, Assumption $\langle 2\rangle$:1 and $\langle 3\rangle$1.
$\langle 3\rangle$3. $\langle [\![Q]\!], M\rangle \overset{(\mathtt{fail},\mathtt{fail})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, M\rangle$, Assumption $\langle 2\rangle$:3 and PT SOS.
$\langle 3\rangle$4. $\langle Q, M\rangle \longrightarrow_\Gamma M$, Assumption $\langle 2\rangle$:2 and $\langle 3\rangle$3.
$\langle 3\rangle$5. $\langle P+Q, M\rangle \longrightarrow_\Gamma M$, from $\langle 3\rangle$2 and $\langle 3\rangle$4 and Γ SOS.
$\langle 3\rangle$6. Q. E. D.

A.1.2 Non-simple programs

Inductive case: sequential compositions.

Leftmost program in sequential composition is applicable.

$\langle 1\rangle$5. Prove $\langle Q \circ P, M\rangle \longrightarrow_\Gamma \langle Q \circ P, N\rangle$ iff
$\langle [\![P]\!] \fatsemi [\![Q]\!], M\rangle \overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P]\!] \fatsemi [\![Q]\!], N\rangle$.
$\langle 2\rangle$1. Assume 1. $\langle P, M\rangle \longrightarrow_\Gamma \langle P, N\rangle$ iff $\langle [\![P]\!], M\rangle \overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P]\!], N\rangle$.
2. $\langle Q \circ P, M\rangle \longrightarrow_\Gamma \langle Q \circ P, N\rangle$.
Prove $\langle [\![P]\!] \fatsemi [\![Q]\!], M\rangle \overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P]\!] \fatsemi [\![Q]\!], N\rangle$.
$\langle 3\rangle$1. $\langle P, M\rangle \longrightarrow_\Gamma \langle P, N\rangle$, Assumption $\langle 2\rangle$:2 and Γ SOS.
$\langle 3\rangle$2. $\langle [\![P]\!], M\rangle \overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P]\!], N\rangle$, Assumption $\langle 2\rangle$:1 and $\langle 3\rangle$1.
$\langle 3\rangle$3. $\langle [\![P]\!] \fatsemi [\![Q]\!], M\rangle \overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P]\!] \fatsemi [\![Q]\!]), N\rangle$, from $\langle 3\rangle$2 and PT SOS.
$\langle 3\rangle$4. Q. E. D.
$\langle 2\rangle$2. Assume 1. $\langle P, M\rangle \longrightarrow_\Gamma \langle P, N\rangle$ iff $\langle [\![P]\!], M\rangle \overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P]\!], N\rangle$.

2. $\langle [\![P]\!] \fatsemi [\![Q]\!], M\rangle \overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P]\!] \fatsemi [\![Q]\!], N\rangle$.

Prove $\langle Q \circ P, M\rangle \longrightarrow_\Gamma \langle Q \circ P, N\rangle$.

$\langle 3\rangle$1. $\langle [\![P]\!], M\rangle \overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P]\!], N\rangle$, Assumption $\langle 2\rangle$:2 and PT SOS.
$\langle 3\rangle$2. $\langle P, M\rangle \longrightarrow_\Gamma \langle P, N\rangle$, Assumption $\langle 2\rangle$:1 and $\langle 3\rangle$1.
$\langle 3\rangle$3. $\langle Q \circ P, M\rangle \longrightarrow_\Gamma \langle Q \circ P, N\rangle$, from $\langle 3\rangle$2 and Γ SOS.
$\langle 3\rangle$4. Q. E. D.

Switch to rightmost program in sequential composition.

$\langle 1\rangle$6. Prove $\langle Q \circ P, M\rangle \longrightarrow_\Gamma \langle Q, M\rangle$ iff $\langle [\![P]\!] \fatsemi [\![Q]\!], M\rangle \overset{(\mathtt{fail},\mathtt{semi})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![Q]\!], M\rangle$

$\langle 2\rangle$1. Assume 1. $\langle P, M\rangle \longrightarrow_\Gamma M$ iff $\langle [\![P]\!], M\rangle \overset{(\mathtt{fail},\mathtt{c})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, M\rangle$, for some c.
2. $\langle Q \circ P, M\rangle \longrightarrow_\Gamma \langle Q, M\rangle$.

Prove $\langle [\![P]\!] \fatsemi [\![Q]\!], M\rangle \overset{(\mathtt{fail},\mathtt{semi})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![Q]\!], M\rangle$.

$\langle 3\rangle$1. $\langle P, M\rangle \longrightarrow_\Gamma M$, Assumption $\langle 2\rangle$:2 and Γ SOS.
$\langle 3\rangle$2. $\langle [\![P]\!], M\rangle \overset{(\mathtt{fail},\mathtt{c})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, M\rangle$, for some c, Assumption $\langle 2\rangle$:1 and $\langle 3\rangle$1.
$\langle 3\rangle$3. $\langle [\![P]\!] \fatsemi [\![Q]\!], M\rangle \overset{(\mathtt{fail},\mathtt{semi})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon \fatsemi [\![Q]\!], M\rangle$, from $\langle 3\rangle$2 and PT SOS.
$\langle 3\rangle$4. $\langle \epsilon \fatsemi [\![Q]\!], M\rangle \overset{(\mathbf{0},\infty)}{\Longrightarrow} \langle [\![Q]\!], M\rangle$, from $\langle 3\rangle$3 and PT SOS.
$\langle 3\rangle$5. $\langle [\![P]\!] \fatsemi [\![Q]\!], M\rangle \overset{(\mathtt{fail},\mathtt{semi})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![Q]\!], M\rangle$,
from $\langle 3\rangle$3 and $\langle 3\rangle$4 and transitivity.
$\langle 3\rangle$6. Q. E. D.

$\langle 2\rangle$2. Assume 1. $\langle P, M\rangle \longrightarrow_\Gamma M$ iff $\langle [\![P]\!], M\rangle \overset{(\mathtt{fail},\mathtt{c})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, M\rangle$, for some c.
2. $\langle [\![P]\!] \fatsemi [\![Q]\!], M\rangle \overset{(\mathtt{fail},\mathtt{semi})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![Q]\!], M\rangle$.

Prove $\langle Q \circ P, M\rangle \longrightarrow_\Gamma \langle Q, M\rangle$.

$\langle 3\rangle$1. $\langle [\![P]\!], M\rangle \overset{(\mathtt{fail},\mathtt{c})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, M\rangle$, for some c, Assumption $\langle 2\rangle$:2 and PT SOS.
$\langle 3\rangle$2. $\langle P, M\rangle \longrightarrow_\Gamma M$, Assumption $\langle 2\rangle$:1 and $\langle 3\rangle$1.
$\langle 3\rangle$3. $\langle Q \circ P, M\rangle \longrightarrow_\Gamma \langle Q, M\rangle$, from $\langle 3\rangle$2 and Γ SOS.
$\langle 3\rangle$4. Q. E. D.

Inductive case: interleaved compositions of non-simple programs. A sequential composition of functions will be reduced in favour of an inapplicable primitive function, if both the sequential composition and the primitive function are in an interleaving composition.

Successful non-simple interleaving.

$\langle 1\rangle$7. Assume P is not simple.
Prove $\langle P + Q, M\rangle \longrightarrow_\Gamma \langle P + Q, N\rangle$ iff

$$\langle [\![P]\!] \,|||\, [\![Q]\!], M\rangle \overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P]\!] \,|||\, [\![Q]\!], N\rangle.$$

$\langle 2\rangle$1. Exactly the same as for simple programs.
$\langle 2\rangle$2. Q. E. D.

Unsuccessful non-simple interleaving.

$\langle 1\rangle$8. Assume P is not simple.
Prove $\langle P + Q, M\rangle \longrightarrow_\Gamma \langle P' + Q, M\rangle$ iff

$$\langle [\![P]\!] \,|||\, [\![Q]\!], M\rangle \overset{(\mathtt{fail},\mathtt{semi})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P']\!] \,|||\, [\![Q]\!], M\rangle, \text{ where } P' \neq P.$$

$\langle 2\rangle$1. Assume 1. $\langle P, M\rangle \longrightarrow_\Gamma \langle P', M\rangle$ iff $\langle [\![P]\!], M\rangle \overset{(\mathtt{fail},\mathtt{semi})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P']\!], M\rangle$.
2. $\langle P + Q, M\rangle \longrightarrow_\Gamma \langle P' + Q, M\rangle$, where $P' \neq P$.

Prove $\langle [\![P]\!] \,|||\, [\![Q]\!], M\rangle \overset{(\mathtt{fail},\mathtt{semi})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P']\!] \,|||\, [\![Q]\!], M\rangle$.

$\langle 3\rangle$1. $\langle P, M\rangle \longrightarrow_\Gamma \langle P', M\rangle$, Assumption $\langle 2\rangle$:2 and Γ SOS.
$\langle 3\rangle$2. $\langle [\![P]\!], M\rangle \overset{(\mathtt{fail},\mathtt{semi})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P']\!], M\rangle$, Assumption $\langle 2\rangle$:1 and $\langle 3\rangle$1.
$\langle 3\rangle$3. $\langle [\![P]\!] \,|||\, [\![Q]\!], M\rangle \overset{(\mathtt{fail},\mathtt{semi})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P']\!] \,|||\, [\![Q]\!], M\rangle$,

from $\langle 3\rangle 2$ and $\mathtt{succ} = \mathtt{semi}$ and PT SOS.

$\langle 3\rangle 4$. Q. E. D.

$\langle 2\rangle 2$. Assume 1. $\langle P, M\rangle \longrightarrow_\Gamma \langle P', M\rangle$ iff $\langle [\![P]\!], M\rangle \overset{(\mathtt{fail},\mathtt{semi})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P']\!], M\rangle$.

2. $\langle [\![P]\!] \,|||\, [\![Q]\!], M\rangle \overset{(\mathtt{fail},\mathtt{semi})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P']\!] \,|||\, [\![Q]\!], M\rangle$.

Prove $\langle P + Q, M\rangle \longrightarrow_\Gamma \langle P' + Q, M\rangle$.

$\langle 3\rangle 1$. $\langle [\![P]\!], M\rangle \overset{(\mathtt{fail},\mathtt{semi})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P']\!], M\rangle$. Assumption $\langle 2\rangle$:2 and PT SOS.

$\langle 3\rangle 2$. $\langle P, M\rangle \longrightarrow_\Gamma \langle P', M\rangle$, Assumption $\langle 2\rangle$:1 and $\langle 3\rangle 1$.

$\langle 3\rangle 3$. $\langle P + Q, M\rangle \longrightarrow_\Gamma \langle P' + Q, M\rangle$, from $\langle 3\rangle 2$ and Γ SOS.

$\langle 3\rangle 4$. Q. E. D.

This concludes the proof. □

A.2 CGP to PT proof

We repeat and prove Proposition 2.

Proposition 11 (Correctness of the translation) *Our translation of CGP into PT is correct. Formally, and writing* $\longrightarrow_{CGP}$ *as the CGP transition relation and* $\Longrightarrow$ *as the PT transition relation, we have:*

$$\forall P, Q, M, N.\quad \begin{array}{ll} \langle P, M\rangle \longrightarrow_{CGP} \langle Q, N\rangle & \textit{iff}\quad \langle [\![P]\!], M\rangle \Longrightarrow^+ \langle [\![Q]\!], N\rangle \wedge \\ \langle P, Q\rangle \longrightarrow_{CGP} M & \textit{iff}\quad \langle [\![P]\!], M\rangle \Longrightarrow^+ \langle \epsilon, M\rangle \end{array}$$

Proof by induction over the structure of the transition relation.

As above, our proof is in two parts. The first part is a proof that the proposition holds for all *simple* CGP programs. Simple programs contain no sequential composition [65]. The second part extends the result to all non-simple programs.

A.2.1 Simple programs

Base case: primitive functions.

Successful primitive function application.

$\langle 1\rangle 1$. Prove $\langle (B, A), M\rangle \longrightarrow_{\mathrm{CGP}} \langle (B, A), N\rangle$ iff

$\langle (B, A, \mathcal{S}_\Gamma)^*, M\rangle \overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle (B, A, \mathcal{S}_\Gamma)^*, N\rangle$

$\langle 2\rangle 1$. Assume $\langle (B, A), M\rangle \longrightarrow_{\mathrm{CGP}} \langle (B, A), N\rangle$.

Prove $\langle (B, A, \mathcal{S}_\Gamma)^*, M\rangle \overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle (B, A, \mathcal{S}_\Gamma)^*, N\rangle$.

$\langle 3\rangle 1$. $\exists \vec{m} \subseteq M.B(\vec{m}) \wedge N = M[A(\vec{m})/\vec{m}]$, Assumption $\langle 2\rangle$ and CGP SOS.

$\langle 3\rangle 2$. $\langle (B, A, \mathcal{S}_\Gamma)^*, M\rangle \overset{(\mathtt{succ},\mathtt{succ})}{\Longrightarrow} \langle \epsilon \overset{\mathtt{succ}}{\triangleright} (B, A, \mathcal{S}_\Gamma)^*, N\rangle$, from $\langle 3\rangle 1$ and PT SOS.

$\langle 3\rangle 3$. $\langle \epsilon \overset{\mathtt{succ}}{\triangleright} (B, A, \mathcal{S}_\Gamma)^*, N\rangle \overset{(\mathbf{0},\infty)}{\Longrightarrow} \langle (B, A, \mathcal{S}_\Gamma)^*, N\rangle$, from PT SOS.

$\langle 3\rangle 4$. $\langle (B, A, \mathcal{S}_\Gamma)^*, M\rangle \overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)}{\Longrightarrow} \langle (B, A, \mathcal{S}_\Gamma)^*, N\rangle$,
from $\langle 3\rangle 2$ and $\langle 3\rangle 3$ and transitivity.

$\langle 3\rangle 5$. Q. E. D.

$\langle 2\rangle 2$. Assume $\langle (B, A, \mathcal{S}_\Gamma)^*, M\rangle \overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle (B, A, \mathcal{S}_\Gamma)^*, N\rangle$.

Prove $\langle (B, A), M\rangle \longrightarrow_{\mathrm{CGP}} \langle (B, A), N\rangle$.

$\langle 3\rangle 1$. $\exists \vec{m} \subseteq M.B(\vec{m}) \wedge N = M[A(\vec{m})/\vec{m}]$, from Assumption $\langle 2\rangle$ and PT SOS.

$\langle 3\rangle 2$. $\langle (B, A), M\rangle \longrightarrow_{\mathrm{CGP}} \langle (B, A), N\rangle$, from $\langle 3\rangle 1$ and CGP SOS.

$\langle 3\rangle 3$. Q. E. D.

Unsuccessful primitive function application.

$\langle 1\rangle 2$. Prove $\langle (B, A), M\rangle \longrightarrow_{\mathrm{CGP}} M$ iff $\langle (B, A, \mathcal{S}_\Gamma)^*, M\rangle \overset{(\mathtt{fail},\mathtt{fail})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, M\rangle$

$\langle 2\rangle 1$. Assume $\langle (B, A), M\rangle \longrightarrow_{\mathrm{CGP}} M$.

Prove $\langle (B, A, \mathcal{S}_\Gamma)^*, M\rangle \overset{(\mathtt{fail},\mathtt{fail})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, M\rangle$.

$\langle 3\rangle 1$. $\neg\exists \vec{m} \subseteq M.B(\vec{m})$, Assumption $\langle 2\rangle$ and CGP SOS.
$\langle 3\rangle 2$. $\langle (B,A,\mathcal{S}_\Gamma)^*, M\rangle \overset{(\texttt{fail},\texttt{fail})}{\Longrightarrow} \langle \epsilon \overset{\texttt{fail}}{\triangleright} (B,A,\mathcal{S}_\Gamma)^*, M\rangle$, from $\langle 3\rangle 1$ and PT SOS.
$\langle 3\rangle 3$. $\langle \epsilon \overset{\texttt{fail}}{\triangleright} (B,A,\mathcal{S}_\Gamma)^*, M\rangle \overset{(\mathbf{0},\infty)}{\Longrightarrow} \langle \epsilon, M\rangle$, from PT SOS.
$\langle 3\rangle 4$. $\langle (B,A,\mathcal{S}_\Gamma)^*, M\rangle \overset{(\texttt{fail},\texttt{fail})(\mathbf{0},\infty)}{\Longrightarrow} \langle \epsilon, M\rangle$, from $\langle 3\rangle 2$ and $\langle 3\rangle 3$ and transitivity.
$\langle 3\rangle 5$. Q. E. D.

$\langle 2\rangle 2$. Assume $\langle (B,A,\mathcal{S}_\Gamma)^*, M\rangle \overset{(\texttt{fail},\texttt{fail})(\mathbf{0},\infty)}{\Longrightarrow} \langle \epsilon, M\rangle$.
Prove $\langle (B,A), M\rangle \longrightarrow_{\text{CGP}} M$.
$\langle 3\rangle 1$. $\neg\exists \vec{m} \subseteq M.B(\vec{m})$, Assumption $\langle 2\rangle$ and PT SOS.
$\langle 3\rangle 2$. $\langle (B,A), M\rangle \longrightarrow_{\text{CGP}} M$, from $\langle 3\rangle 1$ and CGP SOS.
$\langle 3\rangle 3$. Q. E. D.

Inductive case: interleaved compositions of simple programs. This works in the same was as for Γ.

Successful simple interleaving.

$\langle 1\rangle 3$. Prove $\langle P+Q, M\rangle \longrightarrow_{\text{CGP}} \langle P+Q, N\rangle$ iff
$\langle [\![P]\!] \,|||\, [\![Q]\!], M\rangle \overset{(\texttt{succ},\texttt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P]\!] \,|||\, [\![Q]\!], N\rangle$.

$\langle 2\rangle 1$. Assume 1. $\langle P, M\rangle \longrightarrow_{\text{CGP}} \langle P, N\rangle$ iff $\langle [\![P]\!], M\rangle \overset{(\texttt{succ},\texttt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P]\!], N\rangle$.
2. $\langle Q, M\rangle \longrightarrow_{\text{CGP}} \langle Q, N\rangle$ iff $\langle [\![Q]\!], M\rangle \overset{(\texttt{succ},\texttt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![Q]\!], N\rangle$.
3. $\langle P+Q, M\rangle \longrightarrow_{\text{CGP}} \langle P+Q, N\rangle$.
Prove $\langle [\![P]\!] \,|||\, [\![Q]\!], M\rangle \overset{(\texttt{succ},\texttt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P]\!] \,|||\, [\![Q]\!], N\rangle$.
$\langle 3\rangle 1$. $\langle P, M\rangle \longrightarrow_{\text{CGP}} \langle P, N\rangle$ or
$\langle Q, M\rangle \longrightarrow_{\text{CGP}} \langle Q, N\rangle$, Assumption $\langle 2\rangle$:3 and CGP SOS.
$\langle 3\rangle 2$. WLOG, assume $\langle P, M\rangle \longrightarrow_{\text{CGP}} \langle P, N\rangle$.
$\langle 3\rangle 3$. $\langle [\![P]\!], M\rangle \overset{(\texttt{succ},\texttt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P]\!], N\rangle$, Assumption $\langle 2\rangle$:1 and $\langle 3\rangle 2$.
$\langle 3\rangle 4$. $\langle [\![P]\!] \,|||\, [\![Q]\!], M\rangle \overset{(\texttt{succ},\texttt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P]\!] \,|||\, [\![Q]\!]), N\rangle$, from $\langle 3\rangle 3$ and PT SOS.
$\langle 3\rangle 5$. Q. E. D.

$\langle 2\rangle 2$. Assume 1. $\langle P, M\rangle \longrightarrow_{\text{CGP}} \langle P, N\rangle$ iff $\langle [\![P]\!], M\rangle \overset{(\texttt{succ},\texttt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P]\!], N\rangle$
2. $\langle Q, M\rangle \longrightarrow_{\text{CGP}} \langle Q, N\rangle$ iff $\langle [\![Q]\!], M\rangle \overset{(\texttt{succ},\texttt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![Q]\!], N\rangle$
3. $\langle [\![P]\!] \,|||\, [\![Q]\!], M\rangle \overset{(\texttt{succ},\texttt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P]\!] \,|||\, [\![Q]\!], N\rangle$.
Prove $\langle P+Q, M\rangle \longrightarrow_{\text{CGP}} \langle P+Q, N\rangle$.
$\langle 3\rangle 1$. $\langle [\![P]\!], M\rangle \overset{(\texttt{succ},\texttt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P]\!], N\rangle$ or
$\langle [\![Q]\!], M\rangle \overset{(\texttt{succ},\texttt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![Q]\!], N\rangle$, Assumption $\langle 2\rangle$:3 and PT SOS.
$\langle 3\rangle 2$. WLOG, assume $\langle [\![P]\!], M\rangle \overset{(\texttt{succ},\texttt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P]\!], N\rangle$.
$\langle 3\rangle 3$. $\langle P, M\rangle \longrightarrow_{\text{CGP}} \langle P, N\rangle$, Assumption $\langle 2\rangle$:1 and $\langle 3\rangle 2$.
$\langle 3\rangle 4$. $\langle P+Q, M\rangle \longrightarrow_{\text{CGP}} \langle P+Q, N\rangle$, from $\langle 3\rangle 3$ and PT SOS.
$\langle 3\rangle 5$. Q. E. D.

Unsuccessful simple interleaving.

$\langle 1\rangle 4$. Prove $\langle P+Q, M\rangle \longrightarrow_{\text{CGP}} M$ iff $\langle [\![P]\!] \,|||\, [\![Q]\!], M\rangle \overset{(\texttt{fail},\texttt{fail})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, M\rangle$

$\langle 2\rangle 1$. Assume 1. $\langle P, M\rangle \longrightarrow_{\text{CGP}} M$ iff $\langle [\![P]\!], M\rangle \overset{(\texttt{fail},\texttt{fail})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, M\rangle$.
2. $\langle Q, M\rangle \longrightarrow_{\text{CGP}} M$ iff $\langle [\![Q]\!], M\rangle \overset{(\texttt{fail},\texttt{fail})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, M\rangle$.
3. $\langle P+Q, M\rangle \longrightarrow_{\text{CGP}} M$.
Prove $\langle [\![P]\!] \,|||\, [\![Q]\!], M\rangle \overset{(\texttt{fail},\texttt{fail})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, M\rangle$.
$\langle 3\rangle 1$. $\langle P, M\rangle \longrightarrow_{\text{CGP}} M$. Assumption $\langle 2\rangle$:3 and CGP SOS.
$\langle 3\rangle 2$. $\langle [\![P]\!], M\rangle \overset{(\texttt{fail},\texttt{fail})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, M\rangle$, Assumption $\langle 2\rangle$:1 and $\langle 3\rangle 1$.
$\langle 3\rangle 3$. $\langle Q, M\rangle \longrightarrow_{\text{CGP}} M$, Assumption $\langle 2\rangle$:3 and CGP SOS.
$\langle 3\rangle 4$. $\langle [\![Q]\!], M\rangle \overset{(\texttt{fail},\texttt{fail})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, M\rangle$, Assumption $\langle 2\rangle$:2 and $\langle 3\rangle 3$.

$\langle 3\rangle 5.$ $\langle [\![P]\!] \,|||\, [\![Q]\!], M\rangle \overset{(\texttt{fail},\texttt{fail})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, M\rangle$,
from $\langle 3\rangle 2$ and $\langle 3\rangle 4$ and $\texttt{fail} < \texttt{sync}$ and PT SOS.
$\langle 3\rangle 6.$ Q. E. D.

$\langle 2\rangle 2.$ Assume 1. $\langle P, M\rangle \longrightarrow_{\text{CGP}} M$ iff $\langle [\![P]\!], M\rangle \overset{(\texttt{fail},\texttt{fail})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, M\rangle$.
2. $\langle Q, M\rangle \longrightarrow_{\text{CGP}} M$ iff $\langle [\![Q]\!], M\rangle \overset{(\texttt{fail},\texttt{fail})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, M\rangle$.
3. $\langle [\![P]\!] \,|||\, [\![Q]\!], M\rangle \overset{(\texttt{fail},\texttt{fail})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, M\rangle$.
Prove $\langle P + Q, M\rangle \longrightarrow_{\text{CGP}} M$.
$\langle 3\rangle 1.$ $\langle [\![P]\!], M\rangle \overset{(\texttt{fail},\texttt{fail})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, M\rangle$. Assumption $\langle 2\rangle$:3 and PT SOS.
$\langle 3\rangle 2.$ $\langle P, M\rangle \longrightarrow_{\text{CGP}} M$, Assumption $\langle 2\rangle$:1 and $\langle 3\rangle 1$.
$\langle 3\rangle 3.$ $\langle [\![Q]\!], M\rangle \overset{(\texttt{fail},\texttt{fail})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, M\rangle$, Assumption $\langle 2\rangle$:3 and PT SOS.
$\langle 3\rangle 4.$ $\langle Q, M\rangle \longrightarrow_{\text{CGP}} M$, Assumption $\langle 2\rangle$:2 and $\langle 3\rangle 3$.
$\langle 3\rangle 5.$ $\langle P + Q, M\rangle \longrightarrow_{\text{CGP}} M$, from $\langle 3\rangle 2$ and $\langle 3\rangle 4$ and CGP SOS.
$\langle 3\rangle 6.$ Q. E. D.

A.2.2 Non-simple programs

Inductive case: sequential compositions.

Leftmost program in sequential composition keeps going.

$\langle 1\rangle 5.$ Prove $\langle Q \circ P, M\rangle \longrightarrow_{\text{CGP}} \langle Q \circ P, N\rangle$ iff
$\langle [\![P]\!] \fatsemi [\![Q]\!], M\rangle \overset{(\texttt{succ},\texttt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P]\!] \fatsemi [\![Q]\!], N\rangle$.

$\langle 2\rangle 1.$ Assume 1. $\langle P, M\rangle \longrightarrow_{\text{CGP}} \langle P, N\rangle$ iff $\langle [\![P]\!], M\rangle \overset{(\texttt{succ},\texttt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P]\!], N\rangle$.
2. $\langle Q \circ P, M\rangle \longrightarrow_{\text{CGP}} \langle Q \circ P, N\rangle$.
Prove $\langle [\![P]\!] \fatsemi [\![Q]\!], M\rangle \overset{(\texttt{succ},\texttt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P]\!] \fatsemi [\![Q]\!], N\rangle$.
$\langle 3\rangle 1.$ $\langle P, M\rangle \longrightarrow_{\text{CGP}} \langle P, N\rangle$, Assumption $\langle 2\rangle$:2 and CGP SOS.
$\langle 3\rangle 2.$ $\langle [\![P]\!], M\rangle \overset{(\texttt{succ},\texttt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P]\!], N\rangle$, Assumption $\langle 2\rangle$:1 and $\langle 3\rangle 1$.
$\langle 3\rangle 3.$ $\langle [\![P]\!] \fatsemi [\![Q]\!], M\rangle \overset{(\texttt{succ},\texttt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P]\!] \fatsemi [\![Q]\!]), N\rangle$, from $\langle 3\rangle 2$ and PT SOS.
$\langle 3\rangle 4.$ Q. E. D.

$\langle 2\rangle 2.$ Assume 1. $\langle P, M\rangle \longrightarrow_{\text{CGP}} \langle P, N\rangle$ iff $\langle [\![P]\!], M\rangle \overset{(\texttt{succ},\texttt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P]\!], N\rangle$.
2. $\langle [\![P]\!] \fatsemi [\![Q]\!], M\rangle \overset{(\texttt{succ},\texttt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P]\!] \fatsemi [\![Q]\!], N\rangle$.
Prove $\langle Q \circ P, M\rangle \longrightarrow_{\text{CGP}} \langle Q \circ P, N\rangle$.
$\langle 3\rangle 1.$ $\langle [\![P]\!], M\rangle \overset{(\texttt{succ},\texttt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P]\!], N\rangle$, Assumption $\langle 2\rangle$:2 and PT SOS.
$\langle 3\rangle 2.$ $\langle P, M\rangle \longrightarrow_{\text{CGP}} \langle P, N\rangle$, Assumption $\langle 2\rangle$:1 and $\langle 3\rangle 1$.
$\langle 3\rangle 3.$ $\langle Q \circ P, M\rangle \longrightarrow_{\text{CGP}} \langle Q \circ P, N\rangle$, from $\langle 3\rangle 2$ and CGP SOS.
$\langle 3\rangle 4.$ Q. E. D.

Switch to rightmost program in sequential composition.

$\langle 1\rangle 6.$ Prove $\langle Q \circ P, M\rangle \longrightarrow_{\text{CGP}} \langle Q, M\rangle$ iff $\langle [\![P]\!] \fatsemi [\![Q]\!], M\rangle \overset{(\texttt{fail},\texttt{semi})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![Q]\!], M\rangle$

$\langle 2\rangle 1.$ Assume 1. $\langle P, M\rangle \longrightarrow_{\text{CGP}} M$ iff $\langle [\![P]\!], M\rangle \overset{(\texttt{fail},\texttt{c})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, M\rangle$, for some c.
2.$\langle Q \circ P, M\rangle \longrightarrow_{\text{CGP}} \langle Q, M\rangle$.
Prove $\langle [\![P]\!] \fatsemi [\![Q]\!], M\rangle \overset{(\texttt{fail},\texttt{semi})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![Q]\!], M\rangle$.
$\langle 3\rangle 1.$ $\langle P, M\rangle \longrightarrow_{\text{CGP}} M$, Assumption $\langle 2\rangle$:2 and CGP SOS.
$\langle 3\rangle 2.$ $\langle [\![P]\!], M\rangle \overset{(\texttt{fail},\texttt{c})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, M\rangle$,for some c, Assumption $\langle 2\rangle$:1 and $\langle 3\rangle 1$.
$\langle 3\rangle 3.$ $\langle [\![P]\!] \fatsemi [\![Q]\!], M\rangle \overset{(\texttt{fail},\texttt{semi})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon \fatsemi [\![Q]\!], M\rangle$, from $\langle 3\rangle 2$ and PT SOS.
$\langle 3\rangle 4.$ $\langle \epsilon \fatsemi [\![Q]\!], M\rangle \overset{(\mathbf{0},\infty)}{\Longrightarrow} \langle [\![Q]\!], M\rangle$, from PT SOS.
$\langle 3\rangle 5.$ $\langle [\![P]\!] \fatsemi [\![Q]\!], M\rangle \overset{(\texttt{fail},\texttt{semi})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![Q]\!], M\rangle$,
from $\langle 3\rangle 3$ and $\langle 3\rangle 4$ and transitivity.
$\langle 3\rangle 6.$ Q. E. D.

$\langle 2\rangle$2. Assume 1. $\langle P, M\rangle \longrightarrow_{\mathrm{CGP}} M$ iff $\langle [\![P]\!], M\rangle \overset{(\mathtt{fail},\mathtt{c})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, M\rangle$, for some c.
2. $\langle [\![P]\!] \,;\, [\![Q]\!], M\rangle \overset{(\mathtt{fail},\mathtt{semi})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![Q]\!], M\rangle$.
Prove $\langle Q \circ P, M\rangle \longrightarrow_{\mathrm{CGP}} \langle Q, M\rangle$.
$\langle 3\rangle$1. $\langle [\![P]\!], M\rangle \overset{(\mathtt{fail},\mathtt{c})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, M\rangle$, for some c, Assumption $\langle 2\rangle$:2 and PT SOS.
$\langle 3\rangle$2. $\langle P, M\rangle \longrightarrow_{\mathrm{CGP}} M$, Assumption $\langle 2\rangle$:1 and $\langle 3\rangle$1.
$\langle 3\rangle$3. $\langle Q \circ P, M\rangle \longrightarrow_{\mathrm{CGP}} \langle Q, M\rangle$, from $\langle 3\rangle$2 and CGP SOS.
$\langle 3\rangle$4. Q. E. D.

Inductive case: interleaved compositions of non-simple programs. Successful primitive functions are always chosen over reductions of inapplicable non-simple programs. If neither program in an interleaving composition is applicable, then as many left-hand sides of sequential compositions as possible will be synchronously rewritteen (discarded), in preference to discarding a primitive function.

Applicable non-simple interleaving.

$\langle 1\rangle$7. Assume P is not simple.
Prove $\langle P + Q, M\rangle \longrightarrow_{\mathrm{CGP}} \langle P + Q, N\rangle$ iff
$\langle [\![P]\!] \,|||\, [\![Q]\!], M\rangle \overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P]\!] \,|||\, [\![Q]\!], N\rangle$.
$\langle 2\rangle$1. Exactly the same as for simple programs.
$\langle 2\rangle$2. Q. E. D.

Inapplicable non-simple interleaving. We have two cases.

$\langle 1\rangle$8. Assume 1. P is not simple.
2. Q is simple.
Prove $\langle P + Q, M\rangle \longrightarrow_{\mathrm{CGP}} \langle P' + Q, M\rangle$ iff
$\langle [\![P]\!] \,|||\, [\![Q]\!], M\rangle \overset{(\mathtt{fail},\mathtt{semi})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P']\!] \,|||\, [\![Q]\!], M\rangle$, where $P' \neq P$.
$\langle 2\rangle$1. Assume 1. $\langle P, M\rangle \longrightarrow_{\mathrm{CGP}} \langle P', M\rangle$ iff $\langle [\![P]\!], M\rangle \overset{(\mathtt{fail},\mathtt{semi})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P']\!], M\rangle$.
2. $\langle Q, M\rangle \longrightarrow_{\mathrm{CGP}} M$ iff $\langle [\![Q]\!], M\rangle \overset{(\mathtt{fail},\mathtt{fail})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, M\rangle$.
3. $\langle P + Q, M\rangle \longrightarrow_{\mathrm{CGP}} \langle P' + Q, M\rangle$, where $P' \neq P$.
Prove $\langle [\![P]\!] \,|||\, [\![Q]\!], M\rangle \overset{(\mathtt{fail},\mathtt{semi})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P']\!] \,|||\, [\![Q]\!], M\rangle$.
$\langle 3\rangle$1. $\langle Q, M\rangle \longrightarrow_{\mathrm{CGP}} M \wedge$
$\langle P, M\rangle \longrightarrow_{\mathrm{CGP}} \langle P', M\rangle \wedge P' \neq P$, Assumption $\langle 2\rangle$:3 and CGP SOS.
$\langle 3\rangle$2. $\langle [\![P]\!], M\rangle \overset{(\mathtt{fail},\mathtt{semi})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P']\!], M\rangle$, Assumption $\langle 2\rangle$:1 and $\langle 3\rangle$1.
$\langle 3\rangle$3. $\langle [\![Q]\!], M\rangle \overset{(\mathtt{fail},\mathtt{fail})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, M\rangle$, from Assumption $\langle 2\rangle$:2 and $\langle 3\rangle$1.
$\langle 3\rangle$3.$\langle [\![P]\!] \,|||\, [\![Q]\!], M\rangle \overset{(\mathtt{fail},\mathtt{semi})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P']\!] \,|||\, [\![Q]\!], M\rangle$,
from $\langle 3\rangle$2 and $\mathtt{semi} > \mathtt{fail}$ and PT SOS.
$\langle 3\rangle$4. Q. E. D.
$\langle 2\rangle$2. Assume 1. $\langle P, M\rangle \longrightarrow_{\mathrm{CGP}} \langle P', M\rangle$ iff $\langle [\![P]\!], M\rangle \overset{(\mathtt{fail},\mathtt{semi})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P']\!], M\rangle$.
2. $\langle Q, M\rangle \longrightarrow_{\mathrm{CGP}} M$ iff $\langle [\![Q]\!], M\rangle \overset{(\mathtt{fail},\mathtt{fail})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, M\rangle$.
3. $\langle [\![P]\!] \,|||\, [\![Q]\!], M\rangle \overset{(\mathtt{fail},\mathtt{semi})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P']\!] \,|||\, [\![Q]\!], M\rangle$.
Prove $\langle P + Q, M\rangle \longrightarrow_{\mathrm{CGP}} \langle P' + Q, M\rangle$.
$\langle 3\rangle$1. $\langle [\![P]\!], M\rangle \overset{(\mathtt{fail},\mathtt{semi})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P']\!], M\rangle$,
Assumption $\langle 2\rangle$:3 and $\langle 1\rangle$:1 and PT SOS.
$\langle 3\rangle$2. $\langle P, M\rangle \longrightarrow_{\mathrm{CGP}} \langle P', M\rangle \wedge P' \neq P$, Assumption $\langle 2\rangle$:1 and $\langle 3\rangle$1.
$\langle 3\rangle$3. $\langle [\![Q]\!], M\rangle \overset{(\mathtt{fail},\mathtt{fail})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, M\rangle$,
Assumption $\langle 2\rangle$:3 and $\langle 1\rangle$:2 and PT SOS.
$\langle 3\rangle$4. $\langle Q, M\rangle \longrightarrow_{\mathrm{CGP}} M$, Assumption $\langle 2\rangle$:2 and $\langle 3\rangle$3.
$\langle 3\rangle$5. $\langle P + Q, M\rangle \longrightarrow_{\mathrm{CGP}} \langle P' + Q, M\rangle$, from $\langle 3\rangle$2 and $\langle 3\rangle$4 and CGP SOS.
$\langle 3\rangle$6. Q. E. D.

$\langle 1\rangle$9. Assume 1. P is not simple.
2. Q is not simple.

Prove $\langle P+Q, M\rangle \longrightarrow_{\text{CGP}} \langle P'+Q', M\rangle$ iff
$\langle [\![P]\!] ||| [\![Q]\!], M\rangle \overset{(\texttt{fail},\texttt{semi})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P']\!] ||| [\![Q']\!], M\rangle$, where $P' \neq P$ and $Q' \neq Q$.

$\langle 2\rangle$3. Assume 1. $\langle P, M\rangle \longrightarrow_{\text{CGP}} \langle P', M\rangle$ iff $\langle [\![P]\!], M\rangle \overset{(\texttt{fail},\texttt{semi})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P']\!], M\rangle$.
2. $\langle Q, M\rangle \longrightarrow_{\text{CGP}} \langle Q', M\rangle$ iff $\langle [\![Q]\!], M\rangle \overset{(\texttt{fail},\texttt{semi})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![Q']\!], M\rangle$.
3. $\langle P+Q, M\rangle \longrightarrow_{\text{CGP}} \langle P'+Q', M\rangle$, where $P' \neq P$ and $Q' \neq Q$.
Prove $\langle [\![P]\!] ||| [\![Q]\!], M\rangle \overset{(\texttt{fail},\texttt{semi})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P']\!] ||| [\![Q']\!], M\rangle$.

$\langle 3\rangle$1. $\langle P, M\rangle \longrightarrow_{\text{CGP}} \langle P', M\rangle \wedge$
$\langle Q, M\rangle \longrightarrow_{\text{CGP}} \langle Q', M\rangle \wedge P' \neq P \wedge Q' \neq Q$,
Assumption $\langle 2\rangle$:3 and CGP SOS.

$\langle 3\rangle$2. $\langle [\![P]\!], M\rangle \overset{(\texttt{fail},\texttt{semi})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P']\!], M\rangle$, Assumption $\langle 2\rangle$:1 and $\langle 3\rangle$1.

$\langle 3\rangle$3. $\langle [\![Q]\!], M\rangle \overset{(\texttt{fail},\texttt{semi})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![Q']\!], M\rangle$, Assumption $\langle 2\rangle$:2 and $\langle 3\rangle$1.

$\langle 3\rangle$4. $\langle [\![P]\!] ||| [\![Q]\!], M\rangle \overset{(\texttt{fail},\texttt{semi})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P']\!] ||| [\![Q']\!], M\rangle$,
from $\langle 3\rangle$2 and $\langle 3\rangle$3 and $\texttt{semi} < \texttt{sync}$ and PT SOS.

$\langle 3\rangle$5. Q. E. D.

$\langle 2\rangle$4. Assume 1. $\langle P, M\rangle \longrightarrow_{\text{CGP}} \langle P', M\rangle$ iff $\langle [\![P]\!], M\rangle \overset{(\texttt{fail},\texttt{semi})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P']\!], M\rangle$.
2. $\langle Q, M\rangle \longrightarrow_{\text{CGP}} \langle Q', M\rangle$ iff $\langle [\![Q]\!], M\rangle \overset{(\texttt{fail},\texttt{semi})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![Q']\!], M\rangle$.
3. $\langle [\![P]\!] ||| [\![Q]\!], M\rangle \overset{(\texttt{fail},\texttt{semi})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P']\!] ||| [\![Q']\!], M\rangle$,
where $P' \neq P$ and $Q' \neq Q$.
Prove $\langle P+Q, M\rangle \longrightarrow_{\text{CGP}} \langle P'+Q', M\rangle$.

$\langle 3\rangle$1. $\langle [\![P]\!], M\rangle \overset{(\texttt{fail},\texttt{semi})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P']\!], M\rangle$, Assumption $\langle 2\rangle$:3 and PT SOS.
$\langle 3\rangle$2. $\langle P, M\rangle \longrightarrow_{\text{CGP}} \langle P', M\rangle \wedge P' \neq P$, Assumption $\langle 2\rangle$:1 and $\langle 3\rangle$1.
$\langle 3\rangle$3. $\langle [\![Q]\!], M\rangle \overset{(\texttt{fail},\texttt{semi})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![Q']\!], M\rangle$. Assumption $\langle 2\rangle$:3 and PT SOS.
$\langle 3\rangle$4. $\langle Q, M\rangle \longrightarrow_{\text{CGP}} \langle Q', M\rangle$ Assumption $\langle 2\rangle$:2 and $\langle 3\rangle$3.
$\langle 3\rangle$5. $\langle P+Q, M\rangle \longrightarrow_{\text{CGP}} \langle P'+Q', M\rangle$, from $\langle 3\rangle$2 and $\langle 3\rangle$4 and CGP SOS.
$\langle 3\rangle$6. Q. E. D.

This completes the proof. □

A.3 Γ to PT proof

We repeat and prove Proposition 9.

Proposition 12 (Correctness of the translation) *Our translation of* Γ *into PT is correct. Formally, and writing* $\longrightarrow_\Gamma$ *as the* Γ *transition relation and* $\Longrightarrow$ *as the PT transition relation, we have:*

$$\forall P, Q, M, N. \quad \begin{array}{lll} \langle P, M\rangle \longrightarrow_\Gamma \langle Q, N\rangle & \textit{iff} & \langle [\![P]\!], M\rangle \Longrightarrow^+ \langle [\![Q]\!], N\rangle \wedge \\ \langle P, Q\rangle \longrightarrow_\Gamma M & \textit{iff} & \langle [\![P]\!], M\rangle \Longrightarrow^+ \langle \epsilon, M\rangle \end{array}$$

Proof by induction over the structure of the transition relation.

As above, our proof is in two parts. The first part is a proof that the proposition holds for all *simple* Γ programs. Simple Γ programs contain no sequential composition [65]. The second part extends the result to all non-simple programs.

This proof closely mirrors that of Proposition A.1, to which we refer the reader. The difference between the two proofs comes about because primitive function application using $\mathcal{S}_{G1}$ and $\mathcal{S}_{G2}$ require a state synchronisation after every application of a primitive function.

A.3.1 Simple programs

Base case: primitive functions.

Successful primitive function application.

$\langle 1\rangle$**1.** Prove $\langle (B,A),M\rangle \longrightarrow_\Gamma \langle (B,A),N\rangle$ iff
$\langle((B,A,\mathcal{S}_{G1})\S!)^*,(M,M,\emptyset)\rangle \overset{(\mathtt{succ},\mathtt{succ})}{\Longrightarrow}\overset{(\mathbf{0},\infty)^+}{\Longrightarrow}$
$\overset{(\mathtt{fail},\mathtt{succ}+\mathbf{1})}{\Longrightarrow}\overset{(\mathbf{0},\infty)^+}{\Longrightarrow}\langle((B,A,\mathcal{S}_{G1})\S!)^*,(N,N,\emptyset)\rangle$

$\langle 2\rangle$**1.** Assume $\langle (B,A),M\rangle \longrightarrow_\Gamma \langle (B,A),N\rangle$.
Prove $\langle((B,A,\mathcal{S}_{G1})\S!)^*,M\rangle \overset{(\mathtt{succ},\mathtt{succ})}{\Longrightarrow}\overset{(\mathbf{0},\infty)^+}{\Longrightarrow}\overset{(\mathtt{fail},\mathtt{succ}+\mathbf{1})}{\Longrightarrow}\overset{(\mathbf{0},\infty)^+}{\Longrightarrow}$
$\langle((B,A,\mathcal{S}_{G1})\S!)^*,(N,N,\emptyset)\rangle$.

$\langle 3\rangle$**1.** $\exists \vec{m}\subseteq M.B(\vec{m})\wedge N = M[A(\vec{m})/\vec{m}]$, from Assumption $\langle 2\rangle$ and Γ SOS.

$\langle 3\rangle$**2.** $\langle((B,A,\mathcal{S}_{G1})\S!)^*,(M,M,\emptyset)\rangle \overset{(\mathtt{succ},\mathtt{succ})}{\Longrightarrow}$
$\langle(\epsilon\S!)\overset{\mathtt{succ}}{\triangleright}((B,A,\mathcal{S}_{G1})\S!)^*,(M-\vec{m},M,A(\vec{m}))\rangle$,
from $\langle 3\rangle$1 and PT SOS.

$\langle 3\rangle$**3.** $\langle(\epsilon\S!)\overset{\mathtt{succ}}{\triangleright}((B,A,\mathcal{S}_{G1})\S!)^*,(M-\vec{m},M,A(\vec{m}))\rangle \overset{(\mathbf{0},\infty)}{\Longrightarrow}$
$\langle(!\overset{\mathtt{succ}}{\triangleright}((B,A,\mathcal{S}_{G1})\S!)^*,(M-\vec{m},M,A(\vec{m}))\rangle$, from PT SOS.

$\langle 3\rangle$**4.** $\langle(!\overset{\mathtt{succ}}{\triangleright}((B,A,\mathcal{S}_{G1})\S!)^*,(M-\vec{m},M,A(\vec{m}))\rangle \overset{(\mathtt{fail},\mathtt{succ}+\mathbf{1})}{\Longrightarrow}$
$\langle\epsilon\overset{\mathtt{succ}}{\triangleright}((B,A,\mathcal{S}_{G1})\S!)^*,(N,N,\emptyset)\rangle$,
from PT SOS and definition of N.

$\langle 3\rangle$**5.** $\langle\epsilon\overset{\mathtt{succ}}{\triangleright}((B,A,\mathcal{S}_{G1})\S!)^*,(N,N,\emptyset)\rangle \overset{(\mathbf{0},\infty)}{\Longrightarrow}$
$\langle((B,A,\mathcal{S}_{G1})\S!)^*,(N,N,\emptyset)\rangle$, from PT SOS.

$\langle 3\rangle$**6.** $\langle((B,A,\mathcal{S}_{G1})\S!)^*,(M,M,\emptyset)\rangle \overset{(\mathtt{succ},\mathtt{succ})}{\Longrightarrow}\overset{(\mathbf{0},\infty)^+}{\Longrightarrow}\overset{(\mathtt{fail},\mathtt{succ}+\mathbf{1})}{\Longrightarrow}\overset{(\mathbf{0},\infty)^+}{\Longrightarrow}$
$\langle((B,A,\mathcal{S}_{G1})\S!)^*,(N,N,\emptyset)\rangle$,
from $\langle 3\rangle$2, $\langle 3\rangle$3, $\langle 3\rangle$4, $\langle 3\rangle$5.

$\langle 3\rangle$**7.** Q. E. D.

$\langle 2\rangle$**2.** Assume $\langle((B,A,\mathcal{S}_{G1})\S!)^*,(M,M,\emptyset)\rangle \overset{(\mathtt{succ},\mathtt{succ})}{\Longrightarrow}\overset{(\mathbf{0},\infty)^+}{\Longrightarrow}\overset{(\mathtt{fail},\mathtt{succ}+\mathbf{1})}{\Longrightarrow}\overset{(\mathbf{0},\infty)^+}{\Longrightarrow}$
$\langle((B,A,\mathcal{S}_{G1})\S!)^*,(N,N,\emptyset)\rangle$.
Prove $\langle (B,A),M\rangle \longrightarrow_\Gamma \langle (B,A),N\rangle$.

$\langle 3\rangle$**1.** $\exists \vec{m}\subseteq B(\vec{m})\wedge N = M[A(\vec{m})/\vec{m}]$, Assumption $\langle 2\rangle$ and PT SOS.

$\langle 3\rangle$**2.** $\langle (B,A),M\rangle \longrightarrow_\Gamma \langle (B,A),N\rangle$, from $\langle 3\rangle$1 and Γ SOS.

$\langle 3\rangle$**3.** Q. E. D.

Unsuccessful primitive function application.

$\langle 1\rangle$**2.** Prove $\langle (B,A),M\rangle \longrightarrow_\Gamma M$ iff
$\langle((B,A,\mathcal{S}_{G1})\S!)^*,(M,M,\emptyset)\rangle \overset{(\mathtt{fail},\mathtt{fail})}{\Longrightarrow}\overset{(\mathbf{0},\infty)^+}{\Longrightarrow}$
$\overset{(\mathtt{fail},\mathtt{succ}+\mathbf{1})}{\Longrightarrow}\overset{(\mathbf{0},\infty)^+}{\Longrightarrow}\langle\epsilon,(M,M,\emptyset)\rangle$

$\langle 2\rangle$**1.** Assume $\langle (B,A),M\rangle \longrightarrow_\Gamma M$.
Prove $\langle((B,A,\mathcal{S}_{G1})\S!)^*,M\rangle \overset{(\mathtt{fail},\mathtt{fail})}{\Longrightarrow}\overset{(\mathbf{0},\infty)^+}{\Longrightarrow}\overset{(\mathtt{fail},\mathtt{succ}+\mathbf{1})}{\Longrightarrow}\overset{(\mathbf{0},\infty)^+}{\Longrightarrow}\langle\epsilon,M\rangle$.

$\langle 3\rangle$**1.** $\neg\exists \vec{m}\subseteq M.B(\vec{m})$, Assumption $\langle 2\rangle$ and Γ SOS.

$\langle 3\rangle$**2.** $\langle((B,A,\mathcal{S}_{G1})\S!)^*,(M,M,\emptyset)\rangle \overset{(\mathtt{fail},\mathtt{fail})}{\Longrightarrow}$
$\langle(\epsilon\S!)\overset{\mathtt{fail}}{\triangleright}((B,A,\mathcal{S}_{G1})\S!)^*,(M,M,\emptyset)\rangle$, from$\langle 3\rangle$1 and PT SOS.

$\langle 3\rangle$**3.** $\langle(\epsilon\S!)\overset{\mathtt{fail}}{\triangleright}((B,A,\mathcal{S}_{G1})\S!)^*,(M,M,\emptyset)\rangle \overset{(\mathbf{0},\infty)}{\Longrightarrow}$
$\langle(!\overset{\mathtt{fail}}{\triangleright}((B,A,\mathcal{S}_{G1})\S!)^*,(M,M,\emptyset)\rangle$, from PT SOS.

$\langle 3\rangle$**4.** $\langle(!\overset{\mathtt{fail}}{\triangleright}((B,A,\mathcal{S}_{G1})\S!)^*,(M,M,\emptyset)\rangle \overset{(\mathtt{fail},\mathtt{succ}+\mathbf{1})}{\Longrightarrow}$
$\langle(\epsilon\overset{\mathtt{fail}}{\triangleright}((B,A,\mathcal{S}_{G1})\S!)^*,(M,M,\emptyset)\rangle$, from PT SOS.

$\langle 3\rangle$**5.** $\langle\epsilon\overset{\mathtt{fail}}{\triangleright}((B,A,\mathcal{S}_{G1})\S!)^*,(M,M,\emptyset)\rangle \overset{(\mathbf{0},\infty)}{\Longrightarrow}$
$\langle\epsilon,(M,M,\emptyset)\rangle$, from $\langle 3\rangle$4 and PT SOS.

$\langle 3\rangle$**6.** $\langle((B,A,\mathcal{S}_{G1})\S!)^*,(M,M,\emptyset)\rangle \overset{(\mathtt{fail},\mathtt{fail})}{\Longrightarrow}\overset{(\mathbf{0},\infty)}{\Longrightarrow}$
$\overset{(\mathtt{fail},\mathtt{succ}+\mathbf{1})}{\Longrightarrow}\overset{(\mathbf{0},\infty)}{\Longrightarrow}\langle\epsilon,(M,M,\emptyset)\rangle$,
from $\langle 3\rangle$2, $\langle 3\rangle$3, $\langle 3\rangle$4 and $\langle 3\rangle$5

$\langle 3\rangle$**7.** Q. E. D.

$\langle 2\rangle$**2.** Assume $\langle((B,A,\mathcal{S}_{G1})\S!)^*,(M,M,\emptyset)\rangle \overset{(\mathtt{fail},\mathtt{fail})}{\Longrightarrow}\overset{(\mathbf{0},\infty)^+}{\Longrightarrow}$

$\langle (\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+$
$\Longrightarrow \Longrightarrow$
$\langle \epsilon, (M, M, \emptyset)\rangle$.
Prove $\langle (B, A), M\rangle \longrightarrow_\Gamma \langle (B, A), M\rangle$.
$\langle 3\rangle$1. $\neg\exists \vec{m} \subseteq M.B(\vec{m})$, Assumption $\langle 2\rangle$ and PT SOS.
$\langle 3\rangle$2. $\langle (B, A), M\rangle \longrightarrow_\Gamma M$, from $\langle 3\rangle$1 and Γ SOS.
$\langle 3\rangle$3. Q. E. D.

Inductive case: interleaved compositions of simple programs.

Successful simple interleaving.

$\langle 1\rangle$3. Prove $\langle P + Q, M\rangle \longrightarrow_\Gamma \langle P + Q, N\rangle$ iff
$\langle [\![P]\!] \,|||\, [\![Q]\!], (M, M, \emptyset)\rangle$ $\overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow \Longrightarrow}$
$\overset{(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+}{\Longrightarrow \Longrightarrow}$ $\langle [\![P]\!] \,|||\, [\![Q]\!], (N, N, \emptyset)\rangle$.
$\langle 2\rangle$1. Assume 1. $\langle P, M\rangle \longrightarrow_\Gamma \langle P, N\rangle$ iff
$\langle [\![P]\!], (M, M, \emptyset)\rangle$ $\overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow \Longrightarrow}$
$\overset{(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+}{\Longrightarrow \Longrightarrow}$ $\langle [\![P]\!], (N, N, \emptyset)\rangle$.
2. $\langle Q, M\rangle \longrightarrow_\Gamma \langle Q, N\rangle$ iff
$\langle [\![Q]\!], (M, M, \emptyset)\rangle$ $\overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow \Longrightarrow}$
$\overset{(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+}{\Longrightarrow \Longrightarrow}$ $\langle [\![Q]\!], (N, N, \emptyset)\rangle$.
3. $\langle P + Q, M\rangle \longrightarrow_\Gamma \langle P + Q, N\rangle$.
Prove $\langle [\![P]\!] \,|||\, [\![Q]\!], (M, M, \emptyset)\rangle$ $\overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+}{\Longrightarrow \Longrightarrow \Longrightarrow \Longrightarrow}$
$\langle [\![P]\!] \,|||\, [\![Q]\!], (N, N, \emptyset)\rangle$.
$\langle 3\rangle$1. $\langle P, M\rangle \longrightarrow_\Gamma \langle P, N\rangle$ or
$\langle Q, M\rangle \longrightarrow_\Gamma \langle Q, N\rangle$, Assumption $\langle 2\rangle$:3 and Γ SOS.
$\langle 3\rangle$2. WLOG, assume $\langle P, M\rangle \longrightarrow_\Gamma \langle P, N\rangle$.
$\langle 3\rangle$3. $\langle [\![P]\!], (M, M, \emptyset)\rangle$ $\overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+(\mathtt{fail},\mathtt{succ}+\mathbf{1})^+(\mathbf{0},\infty)^+}{\Longrightarrow \Longrightarrow \Longrightarrow \Longrightarrow}$ $\langle [\![P]\!], (N, N, \emptyset)\rangle$,
Assumption $\langle 2\rangle$:1 and $\langle 3\rangle$2.
$\langle 3\rangle$4. $\langle [\![P]\!] \,|||\, [\![Q]\!], (M, M, \emptyset)\rangle$ $\overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+}{\Longrightarrow \Longrightarrow \Longrightarrow}$
$\langle [\![P]\!] \,|||\, [\![Q]\!], (N, N, \emptyset)\rangle$, from $\langle 3\rangle$3 and PT SOS.
$\langle 3\rangle$5. Q. E. D.
$\langle 2\rangle$2. Assume 1. $\langle P, M\rangle \longrightarrow_\Gamma \langle P, N\rangle$ iff
$\langle [\![P]\!], (M, M, \emptyset)\rangle$ $\overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow \Longrightarrow}$
$\overset{(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+(\mathtt{succ},\mathtt{succ})}{\Longrightarrow \Longrightarrow \Longrightarrow}$ $\langle [\![P]\!], (N, N, \emptyset)\rangle$
2. $\langle Q, M\rangle \longrightarrow_\Gamma \langle Q, N\rangle$ iff
$\langle [\![Q]\!], (M, M, \emptyset)\rangle$ $\overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+}{\Longrightarrow \Longrightarrow \Longrightarrow}$ $\langle [\![Q]\!], N\rangle$
3. $\langle [\![P]\!] \,|||\, [\![Q]\!], (M, M, \emptyset)\rangle$ $\overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+}{\Longrightarrow \Longrightarrow \Longrightarrow}$
$\langle [\![P]\!] \,|||\, [\![Q]\!], (N, N, \emptyset)\rangle$.
Prove $\langle P + Q, M\rangle \longrightarrow_\Gamma \langle P + Q, N\rangle$.
$\langle 3\rangle$1. $\langle [\![P]\!], (M, M, \emptyset)\rangle$ $\overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+}{\Longrightarrow \Longrightarrow \Longrightarrow}$ $\langle [\![P]\!], (N, N, \emptyset)\rangle$ or
$\langle [\![Q]\!], (M, M, \emptyset)\rangle$ $\overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+}{\Longrightarrow \Longrightarrow \Longrightarrow}$ $\langle [\![Q]\!], (N, N, \emptyset)\rangle$,
Assumption $\langle 2\rangle$:3 and PT SOS.
$\langle 3\rangle$2. WLOG, assume $\langle [\![P]\!], (M, M, \emptyset)\rangle$ $\overset{(\mathtt{succ},\mathtt{succ})}{\Longrightarrow}$
$\overset{(\mathbf{0},\infty)^+(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+}{\Longrightarrow \Longrightarrow \Longrightarrow}$ $\langle [\![P]\!], (N, N, \emptyset)\rangle$.
$\langle 3\rangle$3. $\langle P, M\rangle \longrightarrow_\Gamma \langle P, N\rangle$, Assumption $\langle 2\rangle$:1 and $\langle 3\rangle$2.
$\langle 3\rangle$4. $\langle P + Q, M\rangle \longrightarrow_\Gamma \langle P + Q, N\rangle$, from $\langle 3\rangle$3 and Γ SOS.
$\langle 3\rangle$5. Q. E. D.

Unsuccessful simple interleaving.

$\langle 1\rangle$4. Prove $\langle P + Q, M\rangle \longrightarrow_\Gamma M$ iff
$\langle [\![P]\!] \,|||\, [\![Q]\!], (M, M, \emptyset)\rangle$ $\overset{(\mathtt{fail},\mathtt{fail})(\mathbf{0},\infty)^+}{\Longrightarrow \Longrightarrow}$

$\langle 2\rangle$1. Assume 1. $\langle P, M\rangle \longrightarrow_\Gamma M$ iff
$\langle [\![P]\!], (M, M, \emptyset)\rangle \overset{(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, (M, M, \emptyset)\rangle$
2. $\langle Q, M\rangle \longrightarrow_\Gamma M$ iff
$\langle [\![Q]\!], (M, M, \emptyset)\rangle \overset{(\mathtt{fail},\mathtt{fail})(\mathbf{0},\infty)^+(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, (M, M, \emptyset)\rangle$.
3. $\langle P + Q, M\rangle \longrightarrow_\Gamma M$.
Prove $\langle [\![P]\!] \,|||\, [\![Q]\!], (M, M, \emptyset)\rangle \overset{(\mathtt{fail},\mathtt{fail})(\mathbf{0},\infty)^+(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, (M, M, \emptyset)\rangle$.
$\langle 3\rangle$1. $\langle P, M\rangle \longrightarrow_\Gamma M$. Assumption $\langle 2\rangle$:3 and Γ SOS.
$\langle 3\rangle$2. $\langle [\![P]\!], (M, M, \emptyset)\rangle \overset{(\mathtt{fail},\mathtt{fail})(\mathbf{0},\infty)^+(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, (M, M, \emptyset)\rangle$, Assumption $\langle 2\rangle$:1 and $\langle 3\rangle$1.
$\langle 3\rangle$3. $\langle Q, M\rangle \longrightarrow_\Gamma M$, Assumption $\langle 2\rangle$:3 and Γ SOS.
$\langle 3\rangle$4. $\langle [\![Q]\!], (M, M, \emptyset)\rangle \overset{(\mathtt{fail},\mathtt{fail})(\mathbf{0},\infty)^+(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, (M, M, \emptyset)\rangle$, Assumption $\langle 2\rangle$:2 and $\langle 3\rangle$3.
$\langle 3\rangle$5. $\langle [\![P]\!] \,|||\, [\![Q]\!], (M, M, \emptyset)\rangle \overset{(\mathtt{fail},\mathtt{fail})(\mathbf{0},\infty)^+(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, (M, M, \emptyset)\rangle$, from $\langle 3\rangle$2 and $\langle 3\rangle$4 and PT SOS.
$\langle 3\rangle$6. Q. E. D.
$\langle 2\rangle$2. Assume 1. $\langle P, M\rangle \longrightarrow_\Gamma M$ iff
$\langle [\![P]\!], (M, M, \emptyset)\rangle \overset{(\mathtt{fail},\mathtt{fail})(\mathbf{0},\infty)^+(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, (M, M, \emptyset)\rangle$.
2. $\langle Q, M\rangle \longrightarrow_\Gamma M$ iff
$\langle [\![Q]\!], (M, M, \emptyset\rangle \overset{(\mathtt{fail},\mathtt{fail})(\mathbf{0},\infty)^+(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, (M, M, \emptyset)\rangle$.
3. $\langle [\![P]\!] \,|||\, [\![Q]\!], (M, M, \emptyset)\rangle \overset{(\mathtt{fail},\mathtt{fail})(\mathbf{0},\infty)^+}{\Longrightarrow} \overset{(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, (M, M, \emptyset)\rangle$.
Prove $\langle P + Q, M\rangle \longrightarrow_\Gamma M$.
$\langle 3\rangle$1. $\langle [\![P]\!], (M, M, \emptyset)\rangle \overset{(\mathtt{fail},\mathtt{fail})(\mathbf{0},\infty)^+(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, (M, M, \emptyset)\rangle$, Assumption $\langle 2\rangle$:3 and PT SOS.
$\langle 3\rangle$2. $\langle P, M\rangle \longrightarrow_\Gamma M$, Assumption $\langle 2\rangle$:1 and $\langle 3\rangle$1.
$\langle 3\rangle$3. $\langle [\![Q]\!], (M, M, \emptyset)\rangle \overset{(\mathtt{fail},\mathtt{fail})(\mathbf{0},\infty)^+(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle \epsilon, (M, M, \emptyset)\rangle$, Assumption $\langle 2\rangle$:3 and PT SOS.
$\langle 3\rangle$4. $\langle Q, M\rangle \longrightarrow_\Gamma M$, Assumption $\langle 2\rangle$:2 and $\langle 3\rangle$3.
$\langle 3\rangle$5. $\langle P + Q, M\rangle \longrightarrow_\Gamma M$, from $\langle 3\rangle$2 and $\langle 3\rangle$4 and Γ SOS.
$\langle 3\rangle$6. Q. E. D.

A.3.2 Non-simple programs

Inductive case: sequential compositions.

Leftmost program in sequential composition keeps going.

$\langle 1\rangle$5. Prove $\langle Q \circ P, M\rangle \longrightarrow_\Gamma \langle Q \circ P, N\rangle$ iff
$\langle [\![P]\!] \fatsemi [\![Q]\!], (M, M, \emptyset)\rangle \overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \overset{(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P]\!] \fatsemi [\![Q]\!], (N, N, \emptyset)\rangle$.
$\langle 2\rangle$1. Assume 1. $\langle P, M\rangle \longrightarrow_\Gamma \langle P, N\rangle$ iff
$\langle [\![P]\!], (M, M, \emptyset)\rangle \overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \overset{(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P]\!], (N, N, \emptyset)\rangle$.
2. $\langle Q \circ P, M\rangle \longrightarrow_\Gamma \langle Q \circ P, N\rangle$.
Prove $\langle [\![P]\!] \fatsemi [\![Q]\!], (M, M, \emptyset)\rangle \overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow}$

$\overset{(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+}{\Longrightarrow}\langle[\![P]\!]\,;[\![Q]\!],(N,N,\emptyset)\rangle$.

$\langle 3\rangle$1. $\langle P,M\rangle \longrightarrow_\Gamma \langle P,N\rangle$, Assumption $\langle 2\rangle$:2 and Γ SOS.

$\langle 3\rangle$2. $\langle[\![P]\!],(M,M,\emptyset)\rangle \overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \overset{(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+}{\Longrightarrow}\langle[\![P]\!],(N,N,\emptyset)\rangle$, Assumption $\langle 2\rangle$:1 and $\langle 3\rangle$1.

$\langle 3\rangle$3. $\langle[\![P]\!]\,;[\![Q]\!],(M,M,\emptyset)\rangle \overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+}{\Longrightarrow}\langle[\![P]\!]\,;[\![Q]\!],(N,N,\emptyset)\rangle$, from $\langle 3\rangle$2 and PT SOS.

$\langle 3\rangle$4. Q. E. D.

$\langle 2\rangle$.2 Assume 1. $\langle P,M\rangle \longrightarrow_\Gamma \langle P,N\rangle$ iff

$\langle[\![P]\!],(M,M,\emptyset)\rangle \overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+(\mathtt{fail},\mathtt{succ}+\mathbf{1})}{\Longrightarrow} \overset{(\mathbf{0},\infty)^+}{\Longrightarrow}\langle[\![P]\!],(N,N,\emptyset)\rangle$.

2. $\langle[\![P]\!]\,;[\![Q]\!],(M,M,\emptyset)\rangle \overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+}{\Longrightarrow}\langle[\![P]\!]\,;[\![Q]\!]),(N,N,\emptyset)\rangle$.

Prove $\langle Q\circ P,M\rangle \longrightarrow_\Gamma \langle Q\circ P,N\rangle$.

$\langle 3\rangle$1. $\langle[\![P]\!],(M,M,\emptyset)\rangle \overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+}{\Longrightarrow}\langle[\![P]\!],(N,N,\emptyset)\rangle$, Assumption $\langle 2\rangle$:2 and PT SOS.

$\langle 3\rangle$2. $\langle P,M\rangle \longrightarrow_\Gamma \langle P,N\rangle$, Assumption $\langle 2\rangle$:1 and $\langle 3\rangle$1.

$\langle 3\rangle$3. $\langle Q\circ P,M\rangle \longrightarrow_\Gamma \langle Q\circ P,N\rangle$, from $\langle 3\rangle$2 and Γ SOS.

$\langle 3\rangle$4. Q. E. D.

Switch to rightmost program in sequential composition.

$\langle 1\rangle$6. Prove $\langle Q\circ P,M\rangle \longrightarrow_\Gamma \langle Q,M\rangle$ iff $\langle[\![P]\!]\,;[\![Q]\!],(M,M,\emptyset)\rangle \overset{(\mathtt{fail},\mathtt{semi})(\mathbf{0},\infty)^+}{\Longrightarrow} \overset{(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+}{\Longrightarrow}\langle[\![Q]\!],(M,M,\emptyset)\rangle$

$\langle 2\rangle$1. Assume 1. $\langle P,M\rangle \longrightarrow_\Gamma M$ iff

$\langle[\![P]\!],(M,M,\emptyset)\rangle \overset{(\mathtt{fail},\mathbf{c})(\mathbf{0},\infty)^+}{\Longrightarrow} \overset{(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+}{\Longrightarrow}\langle\epsilon,(M,M,\emptyset)\rangle$, for some c.

2.$\langle Q\circ P,M\rangle \longrightarrow_\Gamma \langle Q,M\rangle$.

Prove $\langle[\![P]\!]\,;[\![Q]\!],(M,M,\emptyset)\rangle \overset{(\mathtt{fail},\mathtt{semi})(\mathbf{0},\infty)^+}{\Longrightarrow} \overset{(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+}{\Longrightarrow}\langle[\![Q]\!],(M,M,\emptyset)\rangle$.

$\langle 3\rangle$1. $\langle P,M\rangle \longrightarrow_\Gamma M$, Assumption $\langle 2\rangle$:2 and Γ SOS.

$\langle 3\rangle$2. $\langle[\![P]\!],(M,M,\emptyset)\rangle \overset{(\mathtt{fail},\mathbf{c})(\mathbf{0},\infty)^+(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+}{\Longrightarrow}\langle\epsilon,(M,M,\emptyset)\rangle$, for some c, Assumption $\langle 2\rangle$:1 and $\langle 3\rangle$1.

$\langle 3\rangle$3. $\langle[\![P]\!]\,;[\![Q]\!],(M,M,\emptyset)\rangle \overset{(\mathtt{fail},\mathtt{semi})(\mathbf{0},\infty)^+(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+}{\Longrightarrow}\langle\epsilon\,;[\![Q]\!],(M,M,\emptyset)\rangle$, from $\langle 3\rangle$2 and PT SOS.

$\langle 3\rangle$4. $\langle\epsilon\,;[\![Q]\!],(M,M,\emptyset)\rangle \overset{(\mathbf{0},\infty)}{\Longrightarrow}\langle[\![Q]\!],(M,M,\emptyset)\rangle$, from PT SOS.

$\langle 3\rangle$5. $\langle[\![P]\!]\,;[\![Q]\!],(M,M,\emptyset)\rangle \overset{(\mathtt{fail},\mathtt{semi})(\mathbf{0},\infty)^+(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+}{\Longrightarrow}\langle[\![Q]\!],(M,M,\emptyset)\rangle$, from $\langle 3\rangle$3, $\langle 3\rangle$4 and transitivity.

$\langle 3\rangle$6. Q. E. D.

$\langle 2\rangle$2. Assume 1. $\langle P,M\rangle \longrightarrow_\Gamma M$ iff

$\langle[\![P]\!],(M,M,\emptyset)\rangle \overset{(\mathtt{fail},\mathbf{c})(\mathbf{0},\infty)^+}{\Longrightarrow} \overset{(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+}{\Longrightarrow}\langle\epsilon,(M,M,\emptyset)\rangle$, for some c.

2. $\langle[\![P]\!]\,;[\![Q]\!],(M,M,\emptyset)\rangle \overset{(\mathtt{fail},\mathtt{semi})(\mathbf{0},\infty)^+(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+}{\Longrightarrow}\langle[\![Q]\!],(M,M,\emptyset)\rangle$.

Prove $\langle Q\circ P,M\rangle \longrightarrow_\Gamma \langle Q,M\rangle$.

$\langle 3\rangle$1. $\langle[\![P]\!],(M,M,\emptyset)\rangle \overset{(\mathtt{fail},\mathbf{c})(\mathbf{0},\infty)^+(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+}{\Longrightarrow}\langle\epsilon,(M,M,\emptyset)\rangle$, for some c, Assumption $\langle 2\rangle$:2 and PT SOS.

$\langle 3\rangle$2. $\langle P,M\rangle \longrightarrow_\Gamma M$, Assumption $\langle 2\rangle$:1 and $\langle 3\rangle$1.

$\langle 3\rangle$3. $\langle Q\circ P,M\rangle \longrightarrow_\Gamma \langle Q,M\rangle$, from $\langle 3\rangle$2 and Γ SOS.

$\langle 3\rangle$4. Q. E. D.

Inductive case: interleaved compositions of non-simple programs.

Applicable non-simple interleaving composition.

$\langle 1\rangle 7$. Assume P is not simple.
Prove $\langle P+Q, M\rangle \longrightarrow_\Gamma \langle P+Q, N\rangle$ iff
$\langle [\![P]\!] \,|||\, [\![Q]\!], (M,M,\emptyset)\rangle \overset{(\mathtt{succ},\mathtt{succ})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P]\!] \,|||\, [\![Q]\!], (N,N,\emptyset)\rangle$.
$\langle 2\rangle 1$. Exactly the same as for simple programs.
$\langle 2\rangle 2$. Q. E. D.

Inapplicable non-simple interleaving composition.

$\langle 1\rangle 8$. Assume P is not simple.
Prove $\langle P+Q, M\rangle \longrightarrow_\Gamma \langle P'+Q, M\rangle$ iff
$\langle [\![P]\!] \,|||\, [\![Q]\!], (M,M,\emptyset)\rangle \overset{(\mathtt{fail},\mathtt{semi})(\mathbf{0},\infty)^+(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P']\!] \,|||\, [\![Q]\!], (M,M,\emptyset)\rangle$, where $P' \neq P$.
$\langle 2\rangle .1$ Assume 1. $\langle P, M\rangle \longrightarrow_\Gamma \langle P', M\rangle$ iff $\langle [\![P]\!], (M,M,\emptyset)\rangle \overset{(\mathtt{fail},\mathtt{semi})(\mathbf{0},\infty)^+(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P']\!], (M,M,\emptyset)\rangle$.
2. $\langle P+Q, M\rangle \longrightarrow_\Gamma \langle P'+Q, M\rangle$, where $P' \neq P$.
Prove $\langle [\![P]\!] \,|||\, [\![Q]\!], (M,M,\emptyset)\rangle \overset{(\mathtt{fail},\mathtt{semi})(\mathbf{0},\infty)^+(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P']\!] \,|||\, [\![Q]\!], (M,M,\emptyset)\rangle$.
$\langle 3\rangle 1$. $\langle P, M\rangle \longrightarrow_\Gamma \langle P', M\rangle$, Assumption $\langle 2\rangle$:2 and Γ SOS.
$\langle 3\rangle 2$. $\langle [\![P]\!], (M,M,\emptyset)\rangle \overset{(\mathtt{fail},\mathtt{semi})(\mathbf{0},\infty)^+(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P']\!], (M,M,\emptyset)\rangle$, Assumption $\langle 2\rangle$:1 and $\langle 3\rangle 1$.
$\langle 3\rangle 3$. $\langle [\![P]\!] \,|||\, [\![Q]\!], (M,M,\emptyset)\rangle \overset{(\mathtt{fail},\mathtt{semi})(\mathbf{0},\infty)^+(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P']\!] \,|||\, [\![Q]\!], (M,M,\emptyset)\rangle$, from $\langle 3\rangle 2$ and $\mathtt{succ} = \mathtt{semi}$ and PT SOS.
$\langle 3\rangle 4$. Q. E. D.
$\langle 2\rangle 2$. Assume 1. $\langle P, M\rangle \longrightarrow_\Gamma \langle P', M\rangle$ iff
$\langle [\![P]\!], (M,M,\emptyset)\rangle \overset{(\mathtt{fail},\mathtt{semi})(\mathbf{0},\infty)^+(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P']\!], (M,M,\emptyset)\rangle$.
2. $\langle [\![P]\!] \,|||\, [\![Q]\!], (M,M,\emptyset)\rangle \overset{(\mathtt{fail},\mathtt{semi})(\mathbf{0},\infty)^+(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P']\!] \,|||\, [\![Q]\!], (M,M,\emptyset)\rangle$.
Prove $\langle P+Q, M\rangle \longrightarrow_\Gamma \langle P', M\rangle$.
$\langle 3\rangle 1$. $\langle [\![P]\!], (M,M,\emptyset)\rangle \overset{(\mathtt{fail},\mathtt{semi})(\mathbf{0},\infty)^+(\mathtt{fail},\mathtt{succ}+\mathbf{1})(\mathbf{0},\infty)^+}{\Longrightarrow} \langle [\![P']\!], (M,M,\emptyset)\rangle$, Assumption $\langle 2\rangle$:2 and PT SOS.
$\langle 3\rangle 2$. $\langle P, M\rangle \longrightarrow_\Gamma \langle P', M\rangle$, Assumption $\langle 2\rangle$:1 and $\langle 3\rangle 1$.
$\langle 3\rangle 3$. $\langle P+Q, M\rangle \longrightarrow_\Gamma \langle P'+Q, M\rangle$, from $\langle 3\rangle 2$ and Γ SOS.
$\langle 3\rangle 4$. Q. E. D.

This completes the proof. □

Bibliography

[1] J.-M. Andreoli, C. Hankin, and D. Le Métayer, editors. *Coordination programming: mechanisms, models and semantics.* Imperial College Press, London, England, August 1996.

[2] G. Andrews. *Concurrent programming: principles and practice.* Benjamin Cummings, Redwood City, Califormia, U.S.A., 1991.

[3] M. Aono and T. L. Kunii. Botanical tree image generation. *IEEE Computer Graphics and Applications*, 4(5):10–34, May 1984.

[4] K. R. Apt and E.-R. Olderog. *Verification of sequential and concurrent programs.* Springer-Verlag, New York, 1991.

[5] R. J. Back. *On the correctness of refinement steps in program development.* PhD thesis, Department of Computer Science report A-1978-4, Univ. Helsinki, December 1978.

[6] R. J. R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988.

[7] R. J. R. Back. Refinement calculus, part II: parallel and reactive programs. In *Stepwise refinement of distributed systems LNCS 430*, pages 67–93, Berlin, May/Jun 1989. Springer-Verlag.

[8] 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.

[9] 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.

[10] R. J. R. Back and K. Sere. Stepwise refinement of parallel systems. Technical report A64, Dept. of Comp. Sci, Åbo Akademi, Turku, Finland, August 1988.

[11] R. J. R. Back and J. von Wright. Combining angels, demons and miracles in program specification. Technical report A86, Dept. of Comp. Sci, Åbo Akademi, Turku, Finland, September 1989.

[12] R. J. R. Back and J. von Wright. Refinement calculus, part I: sequential nondeterministic programs. In *Stepwise refinement of distributed systems LNCS 430*, pages 42–66, Berlin, May/Jun 1989. Springer-Verlag.

[13] R. J. R. Back and J. von Wright. Statement inversion and strongest precondition. *Science of computer programming*, 20(3):223–251, 1993.

[14] J. C. M. Baeten and C. Verhoef. Concrete process algebra. In *Handbook of logic in computer science*, Oxford, England, 1992. Clarendon Press.

[15] J. C. M. Baeten and W. P. Weijland. *Process algebra.* Cambridge Univ. Press, Cambridge, England, 1990.

[16] J.-P. Banâtre, A. Coutant, and D. Le Métayer. Parallel machines for multiset transformation and their programming style. *Informationstechnik, Oldenbourg Verlag*, 30(2):99–109, 1988.

[17] J.-P. Banâtre and D. Le Métayer. A new computational model and its discipline of programming. Research report 566, INRIA, 78150 Rocquencourt, France, 1986.

[18] J.-P. Banâtre and D. Le Métayer. The Gamma model and its discipline of programming. *Science of computer programming*, 15(1):55–77, November 1990.

[19] J.-P. Banâtre and D. Le Métayer. Programming by multiset transformation. Research report PI 522, IRISA, Rennes, France, March 1990.

[20] J.-P. Banâtre and D. Le Métayer. Programming by multiset transformation. *Communications of the ACM*, 36(1):98–111, January 1993.

[21] H. P. Barendregt. *The lambda calculus, its syntax and semantics*. North Holland, Amsterdam, The Netherlands, 1984.

[22] J. A. Bergstra and J. W. Klop. Fixed point semantics in process algebras. technical report IW 206, CWI, Amsterdam, The Netherlands, 1992.

[23] J. A. Bergstra and J. W. Klop. The algebra of recursively defined processes and the algebra of regular processes. In A Ponse, C. Verhoef, and S. F. M. van Vlijmen, editors, *Algebra of Communicating Processes, proceedings of ACP94*, pages 1–25, Utrecht, The Netherlands, May 1994. Springer Verlag, Berlin.

[24] J. A. Bergstra, J. W. Klop, and J. V. Tucker. Process algebra with asynchronous communication mechanisms. In S. D. Brookes, A. W. Roscoe, and G. Winskel, editors, *Seminar on concurrency, LNCS 197*, pages 76–95. Springer Verlag, Berlin, 1985.

[25] G. Berry and G. Boudol. The chemical abstract machine. In *17th Principles of programming languages*, pages 81–94, San Fransisco, California, January 1990. ACM, New York.

[26] G. Berry and L. Cosserat. The ESTEREL synchronous programming language and its mathematical semantics. In S. D. Brookes, A. W. Roscoe, and G. Winksel, editors, *Seminar on concurrency, LNCS 197*, pages 389–448, Pittsburgh, Pennsylvania, July 1984. Springer-Verlag, Berlin.

[27] G. Boudol. Some chemical abstract machines. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *A decade of concurrency*, pages 76–95. Springer Verlag, Berlin, 1985.

[28] S. Brookes. Full abstraction for a shared variable parallel language. In *Eighth annual IEEE symposium on Logic in Computer Science*, pages 98–109, Montreal, Canada, June 1993. IEEE, Los Alamitos, California.

[29] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. of the ACM*, 31(3):560–599, 1984.

[30] M. Butler. Stepwise refinement of communicating systems. Technical report A147, Dept. of Comp. Sci, Åbo Akademi, Turku, Finland, February 1994.

[31] N. Carriero and D. Gelernter. How to write parallel programs: A guide to the perplexed. *ACM computing surveys*, 21(3):324–357, September 1989.

[32] N. Carriero and D. Gelernter. *How to write parallel programs: a first course*. MIT press, Cambridge, Massachusetts, 1990.

[33] N. Carriero and D. Gerlernter. Applications experience with linda. In *ACM/SIGPLAN Parallel Programming: Experience with Applications, Languages and Systems (PPEALS)*, pages 173–187, New Haven, Connecticut, July 1988. ACM SIGPLAN press.

[34] P. Le Certen and H. Ruiz Barradas. Programmation d'un noyau unix en gamma. technical report 1489, INRIA-Rennes, June 1991.

[35] K. M. Chandy and J. Misra. *Parallel program design: A foundation*. Addison Wesley, Reading, Massachusetts, 1988.

[36] M. R. V. Chaudron and A. C. N. van Duin. A method for the design of parallel algorithms. a case study: solving triangular systems. In *30th Hawaii International Conference on System Sciences*, pages 320–329, Maui, Hawaii, January 1997. IEEE, California.

[37] P. Ciancarini, R. Gorrieri, and G. Zavattaro. An alternative semantics for the parallel operator of the calculus of Gamma programs. In J.-M. Andreoli, C. Hankin, and D. Le Métayer, editors, *Coordination programming: mechanisms, models and semantics*, London, England, August 1996. Imperial College Press.

[38] T. Cooper and N. Wogrin. *Rule-based programming with OPS5.* Kaufmann, San Mateo, 1988.

[39] C. Creveuil. Implementation of Gamma on the connection machine. In J. B. Banâtre and D. Le Métayer, editors, *Research directions in high-level parallel programming languages*, pages 219–229, Mont Saint-Michel, France, June 1991. Springer-Verlag, Berlin.

[40] 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, June 1989.

[41] T. A. Critchley and K. C. Batty. *Open systems: the reality.* Prentice hall, Hemel Hempstead, England, 1993.

[42] J. Darlington, A. J. Field, P. G. Harrison, D. Harper, G. K. Jouret, P. H. J. Kelly, K. M. Sephton, and D. W. Sharp. Structured parallel functional programming. In H. W. Glaser and P. H. Hartel, editors, *3rd Implementation of functional languages on parallel architectures*, pages 31–51, Southampton, England, June 1991. CSTR 91-07, Dept. of Electr. and Comp. Sci, Univ. of Southampton, England.

[43] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. Sharp, and Q. Wu. Parallel programming using skeleton functions. In A. Bode, M. Reeve, and G. Wolf, editors, *5th Parallel architectures and languages Europe (PARLE), LNCS 694*, pages 146–160, Munich, Germany, June 1993. Springer-Verlag, Berlin.

[44] L. Davis. *Handbook of genetic algorithms.* Van Nostrand Reinhold, New York, 1991.

[45] E. de Jong. *Transaction-based programming.* PhD thesis, Rijksuniversiteit Leiden, The Netherlands, 1992.

[46] E. W. Dijkstra. Guarded commands. *CACM*, 18(8):453–457, August 1975.

[47] G. A. Edgar. *Measure, topology and fractal geometry.* Springer-Verlag, New York, 1992.

[48] L. Errington, C. L. Hankin, and T. P. Jensen. Reasoning about gamma programs. In G. L. Burn, S. Gray, and M. Ryan, editors, *Theory and formal methods*, pages 115–125, Isle of Thorns, Chelwood Gate, Sussex, UK, March 1993. Springer-Verlag, Berlin.

[49] A. J. Field and P. G. Harrison. *Functional programming.* Addison Wesley, Reading, Massachusetts, 1988.

[50] W. Fontana and L. W. Buss. 'The arrival of the fittest': towards a theory of biological organisation. *Bulletin of Mathematical Biology*, 56:1–64, 1994.

[51] W. Fontana and L. W. Buss. What would be conserved 'if the tape were run twice'. *Proc. Nat. Acad. Sci.*, 91:757–761, 1994.

[52] W. Fontana, G. Wagner, and L. W. Buss. Beyond digital naturalism. *Artificial Life*, 1:211–227, 1995.

[53] P. Fradet and D. Le Métayer. Structured Gamma. Technical report PI-989, IRISA, France, March 1996.

[54] V. W. Freeh and G. R. Andrews. `fsc`: a Sisal compiler for both distributed- and shard-memory machines. Technical report TR 95-01, Dept. of Comp. Sci, Univ. of Arizona, Tucson, AZ, February 1995.

[55] U. Frisch, D. d'Humiéres, B. Hasslacher, P. Lallemand, Y. Pomeau, and J.-P. Rivet. Lattice gas hydrodynamics in two and three dimensions. *Complex Systems*, 1:649–707, 1987.

[56] M. Fukuda, L. F. Bic, M. B. Dillencourt, and F. Merchant. Intra- and inter-object coordination with MESSENGERS. In P. Ciancarini and C. Hankin, editors, *Proc. Coordination '96 LNCS 1061*, pages 179–196, Cesena, Italy, April 1996. Springer, Berlin.

[57] M. R. Garey and D. S. Johnson. *Computers and intractability.* W. H. Freeman and company, New York, 1979.

[58] D. Gelernter. Generative communication in Linda. *ACM transactions on programming languages and systems*, 7(1):80–112, 1985.

[59] 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.

[60] J.-Y. Girard. Linear logic. *Theoretical computer science*, 50(1):1–102, 1987.

[61] K. Gladitz and H. Kuchen. Parallel implementation of the gamma operation on bags. In *2nd International Symp. on Parallel Symbolic Computation (PASCO), Lecture Notes Series in Computing vol. 5*, pages 154–163. World Scientific, Singapore, 1994.

[62] 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, September 1994.

[63] D.E. Goldberg. *Genetic Algorithms in search, optimization and machine learning.* Addison-Wesley, 1989.

[64] C. L. Hankin. *Lambda calculi.* Clarendon Press, Oxford, UK, 1994.

[65] C. L. Hankin, D. le Métayer, and D. Sands. A calculus of Gamma programs. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Fifth international workshop on languages and compilers for parallel computing*, pages 342–355, New Haven, CT, USA, August 1992. Springer-Verlag, Berlin.

[66] 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, September 1992. BIGRE, 81-82.

[67] C. L. Hankin, D. le Métayer, and D. Sands. A parallel programming style and its algebra of programs. In A. Bode, M. Reeve, and G. Wolf, editors, *5th Parallel architectures and languages Europe (PARLE), LNCS 694*, pages 367–378, Munich, Germany, June 1993. Springer-Verlag, Berlin.

[68] Dan Heller. *XView programming manual.* O'Reilly and Associates, Sebastopol, California, 1989.

[69] C. A. R. Hoare. *Communication sequential processes.* Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[70] H. Hoffmann. Implementierung von Gamma auf MIMD-Rechnern mit verteiltem Speicher. Master's thesis, Lehrstuhl fi Informatik II, RWTH-Aachen, Germany, February 1996.

[71] C. J. Hogger. *Essentials of logic programming.* Clarendon, Oxford, England, 1990.

[72] J.H. Holland. *Adaptation in Natural and Artificial Systems.* Univ. Michigan, Ann Arbor, 1975.

[73] A. Huth and C. Wissel. The simulation of movement of fish schools. *J. Theoretical Biology*, 156(3):365–385, 1992.

[74] K. Culik II and J. Kari. Mechanisms for pattern formation. *Complex Systems*, 7:347–365, 1993.

[75] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[76] H.-M. Järvinen. *The design of a specification language for reactive systems.* PhD thesis, Tampere Univ. Technology, Finland, April 1992.

[77] H.-M. Järvinen, R. Kurki-Suonio, M. Sakkinen, and K. Systä. Object-orientation specification of reactive systems. In *International conference on software engineering*, pages 63–71, Nice, France, March 1990. IEEE Computer Society Press, Los alamitos, California.

[78] F. P. Brooks Jr. *The mythical man-month, 25th Anniversary edition.* Addison Wesley, Reading, Massachusetts, 1995.

[79] J. A. Kaandorp. *Fractal modelling: growth and form in biology.* Springer-Verlag, Berlin, 1994.

[80] K. Kassner. Sidebranching in noiseless diffusion-limited aggregation. *Fractals*, 1(2):205–228, 1993.

[81] P. H. J. Kelly. *Functional programming for loosely-coupled multiprocessors.* Pitman publishing, London, England, 1989.

[82] B. W. Kernighan and D. W. Ritchie. *The C programming language - ANSI C.* Prentice Hall, Englewood Cliffs, New Jersey, second edition edition, 1988.

[83] S. Kirkpatrick, C.D. Gelatt jr., and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

[84] D. E. Knuth. *The art of computer programming, volume 2: Seminumerical algorithms.* Addison Wesley, Reading, Massachusetts, second edition, 1980.

[85] R. Kurki-Suonio. Operational specification with joint actions: serializable databases. *Distributed computing*, 6:19–37, 1992.

[86] R. Kurki-Suonio. Stepwise design of real-time systems. *IEEE transactions on software engineering*, 19(1):56–69, 1993.

[87] R. Kurki-Suonio, K. Systä, and Jüri Vain. Real-time specification and modelling with joint actions. *Science of computer programming*, 20:113–140, 1993.

[88] L. Lamport. What good is temporal logic? In *Information processing '83*, pages 657–668, Paris, France, September 1983. North Holland, Amsterdam.

[89] L. Lamport. How to write a proof. Technical report 94, Digital Equipment Systems Research center, Palo Alto, California, February 1993.

[90] D. H. Lehmer. Mathematical methods in large-scale computing units. In *Proc. 2nd symp. large-scale digital calculating machinery*, pages 141–146, Cambridge, Massachusetts, 1951. Harvard University press.

[91] A. Lindenmayer. Mathematical models for cellular interactions in development. *Theoretical Biology*, 18:280–299, 1968.

[92] B. Lubachevsky. Efficient parallel simulations of asynchronous cell arrays. *Complex systems*, 1:1099–1123, 1987.

[93] B. Lubachevsky. Efficient parallel simulations of dynamic Ising spin systems. *Journal of computational physics*, 75:103–122, 1988.

[94] B. B. Mandelbrot and T. Vicsek. Directed recursive models of fractal growth. *J. Physics*, 22:377–381, 1989.

[95] H. McEvoy. Gamma, chromatic typing and vegetation. In J.-M. Andreoli and C. Hankin, editors, *Coordination programming: mechanisms, models and semantics*, London, England, August 1996. Imperial College Press.

[96] H. McEvoy. Context sensitivity and synchronisation as taxonomics for parallel programming. In *30th Hawaii International Conference on System Sciences*, pages 369–378, Maui, Hawaii, January 1997. IEEE, California.

[97] H. McEvoy and P. H. Hartel. Local linear logic for locality consciousness in multiset transformation. In M. Hermenegildo and S. D. Swierstra, editors, *7th Programming languages: implementations, logics and programs (PLILP), LNCS 982*, pages 357–379, Utrecht, The Netherlands, September 1995. Springer-Verlag, Berlin.

[98] H. McEvoy and J. Kaandorp. On modelling environmentally-sensitive growth forms and cellular automata using multiset transformation. *Fractals*, 4(4), 1997.

[99] E. Mendelson. *Introduction to Mathematical Logic, 2nd edition.* Wadsworth and Brooks/Cole, Monterey, California, 1987.

[100] D. Le Métayer. Higher-order multiset processing. In G. E. Blelloch, K. M. Chandy, and S. Jagannathan, editors, *Specification of Parallel Languages, DIMACS 18*, pages 179–200. American Mathematical Society, May 1994.

[101] R. Milner. *A calculus of communicating systems, LNCS 92.* Springer Verlag, Berlin, 1980.

[102] R. Milner. *Communication and concurrency.* Prentice Hall, Hemel Hempstead, England, 1989.

[103] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part 1. Technical report ECS-LFCS-89-85, Univ. Edinburgh, 1985.

[104] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part 2. Technical report ECS-LFCS-89-86, Univ. Edinburgh, 1985.

[105] F. Moller. Axioms for concurrency. Technical report CST-59-89, Univ. Edinburgh, 1989.

[106] C. Morgan. The specification statement. *ACM Transactions on programming languages and systems*, 10(3):403–419, 1988.

[107] J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of computer programming*, 9:287–306, 1987.

[108] V. K. Murthy and E. V. Krishnamurthy. Probabilistic parallel programming based on multiset transformation. *Future generation computer systems (FGCS)*, 11(3):283–293, June 1995.

[109] R. De Nicola and M. Hennessy. CCS without τ's. In H. Ehrig, R. Kowalski, G. Levi, and U. Montanari, editors, *Theory and practise of software development '87 (TAPSOFT)*, pages 138–152, Pisa, Italy, March 1987. Springer-Verlag, Berlin.

[110] H. R. Nielson and F. Nielson. *Semantics with applications: A formal introduction.* John Wiley & Sons, Chichester, England, 1991.

[111] L. Niemeyer, L. Pietronero, and H. J. Weismann. Fractal dimension of dielectric breakdown. *Physical review letters*, 52(12):1033–1036, March 1984.

[112] L. C. Paulson. *ML for the working programmer.* Cambridge university press, Cambridge, England, 1991.

[113] S. L. Peyton Jones. Parallel implementations of functional programming languages. *The Computer Journal*, 32(2):175–186, 1989.

[114] G. D. Plotkin. A structural approach to operational semantics. Technical report DAIMI FN-19, Dept. of Comp. Sci, Aarhus Univ., Denmark, September 1981.

[115] P. Prusinkiewicz, M. Hammel, J. Hanan, and R. Měch. Visual models of plant development. In *Handbook of formal languages*, Berlin, 1996 (to appear). Springer-Verlag.

[116] P. Prusinkiewicz, J. Hanan, and A. Lindenmayer. *Lindenmayer systems, fractals and plants. Lecture notes in biomathematics 79.* Springer-Verlag, New York, 1989.

[117] P. Prusinkiewicz, M. James, and R. Měch. Synthetic topiary. *Computer Graphics*, pages 351–358, 1994. SIGGRAPH '94 Proceedings.

[118] P. Prusinkiewicz and A. Lindenmayer. *The algorithmic beauty of plants.* Springer, New York, 1990.

[119] Å. Wikström. *Functional programming using Standard ML.* Prentice Hall, Hemel Hempstead, England, 1987.

[120] S.M. Ross. *A course in simulation.* Macmillan, 1991.

[121] A. I. T. Rowstron and A. M. Wood. BONITA: a set of tuple space primitives for distributed coordination. In *30th Hawaii International Conference on System Sciences*, pages 379–398, Maui, Hawaii, January 1997. IEEE, California.

[122] G. Rozenberg and A. Salomaa. *The mathematical theory of L-systems.* Academic press, New York, 1980.

[123] G. Rozenberg and A. Salomaa. *The book of L.* Springer, Berlin, 1986.

[124] A. Salomaa. *Formal languages.* Academic press, New York, 1973.

[125] A. Salomaa. *Jewels of formal language theory.* Pitman, London, 1981.

[126] D. Sands. A compositional semantics of combining forms for Gamma programs. In D. Bjørner, M. Broy, and I. V. Pottosin, editors, *Formal methods in programming and their applications, LNCS 735*, pages 43–56, Academgorodok,Novosibirsk,Russia, Jun/Jul 1993. Springer Verlag, Berlin.

[127] D. Sands. Composed reduction systems. In J.-M. Andreoli, C. Hankin, and D. Le Métayer, editors, *Coordination programming: mechanisms, models and semantics*, London, England, August 1996. Imperial College Press.

[128] V.A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proc. of the Seventeenth ACM Symposium on Principles of Programming Languages*, pages 232–245. ACM, New York, 1990.

[129] H. G. Schlegel. *Allgemeine Mikrobiologie.* Georg Thieme Verlag, Stuttgart, Germany, 1976.

[130] B. Schoïifisch and K. Hadeler. Dimer automata and cellular automata. *Physica D*, 94:188–204, 1996.

[131] K. Sere. Communication in processor farms: a case study in reactive systems refinement. Technical report A122, Dept. of Comp. Sci, Åbo Akademi, Turku, Finland, January 1991.

[132] J. Siegel. *CORBA fundamentals and programming.* John Wiley and Sons, Inc, New York, U.S.A., 1996.

[133] P. M. A. Sloot, J. M. Voogt, D. de Kanter, and L. O. Hertzberger. Simulated annealing: comparison of vector and parallel implementations. Technical report CS-93-06, Univ. of Amsterdam, The Netherlands, October 1993.

[134] S. Thompson. *Miranda: the craft of functional programming.* Addison Wesley, Harlow, England, 1995.

[135] S. Thompson. *Haskell: the craft of functional programming.* Addison Wesley, Harlow, England, 1996.

[136] T. Toffoli and N. Margolus. *Cellular automata machines.* MIT press, Cambridge, Massachusetts, 1991.

[137] R. Tolksdorf. Coordinative applications, structured coordination and meta coordination. In *30th Hawaii International Conference on System Sciences*, pages 391–392, Maui, Hawaii, January 1997. IEEE, California.

[138] S. Ulam. On the Monte Carlo method. In *Proc. 2nd symp. on large-scale digital calculating machinery*, pages 207–212, Cambridge, Mass., September 1951. Harvard University press.

[139] M. Vieillot. Premiers pas de Gamma avec une PAM. Rapport de stage, IFSIC, IRISA, Univ. de Rennes, France, 1992.

[140] R. F. Voss. Birth, death, step size and the shape of DLA. *Fractals*, 1(2):141–147, 1993.

[141] L. Wall. *Programming Perl, second edition*. O'Reilly and Associates, Sebastopol, California, 1996.

[142] R. A. Whiteside and J. S. Leichter. Using Linda for supercomputing on a local area network. In *Supercomputing '88*, pages 192–199, Orlando, Florida, November 1988. IEEE and ACM SIGARCH.

[143] T. A. Witten and L. M. Sander. Diffusion-limited aggregation. *Physical review B*, 27(9):5686–5697, 1984.

[144] L. Wittgenstein. *Philosophical Investigations*. Basil Blackwell, Oxford, England, 1958.

[145] S. Wolfram. *Cellular automata and complexity*. Addison Wesley, Reading, Massachusetts, 1994.