CHAPTER 1

Introduction

CHAPTER 2

Background

This chapter reviews the many sources that I have relied upon in designing MIN, and also some of the apparently relevant ideas that I have rejected. There are at least three easily distinguishable communities within computer science that study the semantics of programs with a view to optimising or otherwise transforming them. Any of these could plausibly be the starting point for an understanding of the semantics of multi-threaded programs. They are:

- the community which, back in the 1980s, worked out how to write almost perfect C compilers. This community likes to think about the kind of problem where a program is presented in a form similar to assembler, such as a register transfer language or a single static assignment form, and needs to be converted into a similar form, such as machine code for a particular processor. Areas of current research include code compression and the interaction of instruction re-ordering and register allocation.
- the functional programming community which, in the 1990s, worked out how to compile functional programs almost as well as C programs, even though they are an abstraction far from physical machines, and do not offer programmers much opportunity to direct the implementation and optimisation of their code. Areas of current research include weird and wonderful analyses that deduce that parts of a program are only used in certain ways, thus allowing a specialised and more efficient implementation.
- the process calculus community, which has the unique advantage of truly understanding concurrency, but which has never really grappled with real-world problems, and which has never produced a programming language that is more than a toy. Current areas of research include the invention of new calculi and the understanding of the semantic effect of decisions in their design.

I have almost exclusively drawn on the work of the third of these communities. Though the other two appear to provide promising starting points, I believe that there are significant obstacles to understanding the problems of multi-threaded programming as perturbations of ordinary single-threaded imperative or functional programming, and that knuckling down and working out how to do real programming with process calculi is a line of attack that is much more likely to succeed.

Having said that, the task of programming with process calculi is a great deal easier thanks to the possibility of imitating the historical development of functional languages. We won't have to do nearly as much trial and error. Many of the famous failures of functional programming, such as the idea of compiling programs using the S and K combinators, have close analogies in process calculi. While the bulk and the detail of my work draws on process calculi, the large-scale structure of it comes from functional programming.

There is a fourth source which I did not include in the above list because there is currently very little work on its applications to programming, and that is the literature on Interaction Nets. Interaction Nets have contributed enormously to my work, in notation, intuition and nomenclature. This beautiful model of computation arose from the study of proofs and logic, but its attractions to me relative to other models of similar power and simplicity are that it is linear and that it is graphical. It also enjoys some good mathematical properties. Finally, but beyond the scope of this thesis, staring back from the page are user-defined rewrite rules that would map beautifully onto machine code. The IN in MIN stands for Interaction Nets (the M stands for Multi-threaded).

MIN can therefore be summarised as a marriage of Interaction Nets with the mathematics of process calculi, guided by experience from functional programming.

The structure of this chapter is as follows. In section 2.1 I review some of the literature on Interaction Nets. In section 2.2 I give an approximately historical account of the techniques from process calculi that I have used. My type system, which is supposed be competent but uninspiring, does not deserve a separate section of its own; all the required techniques are contained in sections 6.2 to 6.7 of [**33**], for example. Functional programming is a sufficiently mature field that I can point you at the 1987 textbook that I used [**27**] and supplement it with a brief explanation, in section 2.5, of how I have applied the principal ideas.

2.1. Interaction Nets

Yves Lafont introduced Interaction Nets in 1990 [**16**], but if you read one paper it should be his 1997 paper on Interaction Combinators [**17**]. For the last few years the Interaction Net torch has been carried mainly by Ian Mackie and Maribel Fernández. Their project like mine is to use Interaction Nets as an intermediate language in a compiler, but whereas I am interested in multi-threaded languages they are interested primarily in functional languages, and in particular in sharing. Their work and mine therefore have both a healthy overlap and an interesting difference of perspective.

2.1.1. Interaction Nets. An interaction net version of the AND gate example that I used to illustrate MIN in chapter 1 is shown in figure 2.1.1. The comparison with the MIN version is instructive.

One obvious difference is a superficial difference of notation. The figure uses a notation in which nodes are drawn as circles and the principal port of each node is indicated with an arrow. This is the notation used in [**10**], for example. A triangle-based notation similar to that which I use for MIN is also sometimes used for Interaction Nets.

2.1.1.1. *No linking.* Two important differences between the MIN and interaction net version of figure 2.1.1 are actually aspects of a single fundamental difference of approach. First, the interaction net treats all four node symbols on a par, whereas the MIN program partitions them into two constructor symbols and two destructor symbols. Second, the interaction net doesn't actually have a net (which in MIN would be called the main graph); it only has an alphabet (which in MIN would be called the node declarations) and some rewrite rules.

What is going on is as follows. Diagram 2.1.1 does not define a self-contained communicating program like the MIN program does. Instead, it defines an *interaction net system*, a language which can be used for writing programs. The phrase 'Interaction Nets' refers to the collection of all possible such systems, and a *net*

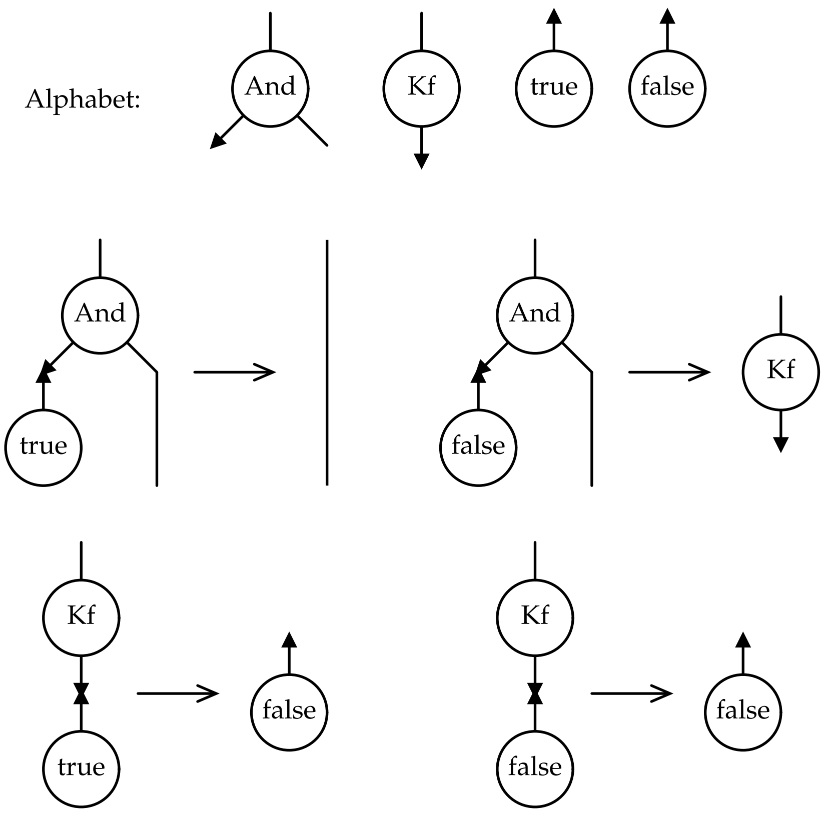

FIGURE 2.1.1. An interaction net implementation of an AND gate. Compare the MIN version in chapter 1.

is a program which makes sense only in the context of a particular system. For example, a single 'and' node would be a net of the system in figure 2.1.1 which represents a single one-shot AND gate.

This might look like playing with words but it is far from trivial. It is true that the node declarations and rewrite rules could be included in every interaction net to make them look like MIN programs, and that the node declarations and rewrite rules of many MIN programs can be combined to look like an interaction net system. However, these tricks only take care of the operational semantics. MIN also has a linking operation, which allows programs with different node symbols and rewrite rules to be joined together. Interaction nets do not, so nets can only be composed to form larger nets if they belong to the same system.

The name-space of node symbols for an interaction net is therefore finite and global, a property only of the interaction net system and not of any particular net. Any distinction made between constructors and destructors would be of no consequence. This explains why Interaction Nets make no such distinction. In contrast, MIN's linking operation treats constructors and destructors differently, and programmers have good reason to care which is which.

The impact of linking on the theory is enormous. In the absence of linking, the set of contexts which can be used to try to observe the differences between programs is considerably smaller, and this means that the equivalence relation on programs can be larger. For example, in the system in figure 2.1.1 it is not possible to distinguish 'true' from 'false', because there is no 'if' destructor symbol. MIN's linking operation provides a way of introducing one.

The definition of linking and recognition of its importance are among my novel contributions.

2.1.2. Observational equivalence. My theory of observational equivalence for MIN programs, which I develop in chapter 3, is based on the concept of simulation, and imitates the theory developed for CCS, the π-calculus, and similar process calculi, which I review in section 2.2. Fernández and Mackie also have a simulation-based theory of observational equivalence, for Interaction Nets, which imitates theory developed for the lambda calculus and for functional programming languages [**10**]. Comparing their project with mine, one would expect this to be the main area of overlap, and it is.

Encouragingly, the end result is exceedingly similar. I believe that if their definition of the observational equivalence relation were applied to a MIN program that is an interaction net, it would produce exactly the same relation as my definition. However, there are plenty of differences in the detail.

Like mine, their definition involves partitioning the node symbols into constructors and destructors, which is fascinating given that there is no obvious basis for such a distinction in Interaction Nets. I think this situation has arisen because they imitated theory developed for functional programming languages which *do* have a concept of linking and hence a distinction between constructors and destructors. The necessity of making a distinction is thus an evolutionary relic which they have inherited without inheriting the evolutionary pressure which shaped it.

Whatever its origins, this feature of their definition is exceedingly odd. The partition chosen does not affect the operational semantics of nets in any way, and yet it makes a large difference to the observational equivalence relation. At one extreme it is possible to choose all node symbols to be constructors, in which case the equivalence relation is extremely fine: it merely relates all non-interacting programs and includes the reaction relation. At the other extreme it is possible to choose all node symbols to be destructors, in which case the equivalence relation is the universal one.

Their definition differs from mine in its labelled transition relation. My labelled transition relation allows input transitions to occur at any time at any port, whereas theirs is rather more efficient and gives criteria for eliminating most input transitions. On the other hand, my labelled transitions only ever involve a single port, whereas they have some that join two ports with a wire, and even some which join a tuple of ports to a node. I believe that the minimal labelled transition relation will lie in the intersection of theirs and mine. However, I think mine is the minimal labelled transition system that does not require a special definition of simulation (one that differs from that used for CCS).

The observation and proof that labels involving more than one port are unnecessary is related to the absence of a 'match' primitive in Interaction Nets and in MIN, and is one of my novel contributions.

2.1.3. Interaction combinators. A particularly interesting interaction net system is that of Lafont's *interaction combinators* [**17**]. The system has just three node symbols, but is able to encode any other interaction net system. The definition of 'encode' used to make this statement is impressively strict. The encoding algorithm is compositional: it replaces each node of a net belonging to the other system with a net made of interaction combinators, and wires those nets together exactly as the nodes were wired together. If the original net has no redexes, then neither does its encoding. Otherwise, performing any reaction in the original net and then encoding it yields the same result as encoding the original net and then performing a finite number of reactions.

I am not aware of any satisfactory observational equivalence relation for nets of interaction combinators. Interaction combinators are completely deterministic, like all interaction net systems, and so the reaction relation generates a sound equivalence relation. There are some well known additional equivalences that intuitively should also hold for interaction combinators [**10**]. Lafont defines a 'path equivalence' relation, which can be axiomatised by a subset of these. The smallest equivalence relation containing all of them is larger. It is usable for simple optimisations, but it is not observational, and is rather uninformative.

My theory cannot be applied to interaction combinators because there is no good way of partitioning the three node symbols into constructors and destructors. [**10**] applies its theory, which is similar to mine, with all three symbols as destructors, but unfortunately the resulting equivalence relation relates any two nets with the same number of free ports.[1] Lafont points out that it is possible to double the number of interaction combinators by making a constructor and a destructor version of each, and that his encoding has the property that all redexes that arise involve one constructor and one destructor. This change allows my theory to be applied. However, applying my theory to that system yields an equivalence relation that is too fine: it does not even contain Lafont's path equivalence relation. This is because my theory accounts for the effect of linking, which is inappropriate.

The problem remains open. Programs written using Lafont's interaction combinators are not MIN programs.

2.1.4. Multiple principal ports. All node symbols in Interaction Nets have exactly one principal port. Since all redexes consist of two nodes joined only by their principal ports, it is impossible for two redexes to overlap, and so all interaction net systems have a strong 'one-step diamond' confluence property. Thus they are examples of languages that are concurrent but deterministic.

The theory of a deterministic language is exceptionally simple, and that is one of the attractive features of Interaction Nets, but for real programming determinism is not always a good thing. As I explained in chapter 1, MIN allows destructors to have more than one principal port, in which case they are called arbiters. A similar idea appears in [**8**], but without the useful restriction to destructors.

That paper also shows that all arbiters can be encoded using a single arbiter called 'amb', which reacts to form one of two wirings depending on at which of its two principal ports it reacts. It seems to be quite hard to implement general arbiters well, so an encoding of this sort in terms of a particular arbiter may well be a pragmatic approach to implementing MIN. The particular arbiter I favour is not 'amb' but 'merge' (similar to 'angelic merge' in [**8**]) of which more in chapter 6.

The 'multiple principal ports' approach is not the only way of introducing non-determinism into Interaction Nets that is found in the literature. A particularly dumb suggestion is to allow multiple rewrite rules with the same redex. This introduces all the down-sides of non-determinism but does not achieve arbitration. A better idea is to allow wires to fork, but this unnecessarily sacrifices linearity and

[1]This was a bit of a shock to Ian Mackie when I alerted him to it. Having discussed it we still think it is true. In particular, the paper has no negative results, and its final example is especially worrying.

a certain amount of information hiding. A fourth approach is to use hyper-edges that have more than two 'ends'. I have not investigated this approach in detail, but in other graphical formalisms hyper-edges can simply be viewed as nodes, with their several ports joined by linear wires to the things explicitly called nodes.

One of my novel contributions is the observation that my simulation-based observational theory generalises flawlessly to the case of destructors with multiple principal ports. This is really quite an unexpected result, and a very pleasant one.

2.1.5. Types. Type technology is remarkably portable between programming languages and consequently there is a healthy array of type systems for Interaction Nets, beginning with one in Lafont's original paper. That Interaction Nets arose from a fragment of linear logic also provides excellent intuitive support for the design of a type system.

Most type systems for Interaction Nets, including that in Lafont's original paper, have in common that every port is annotated both with a type and with a polarity. The polarity can be 'input' or 'output', and two ports can only be joined if one is an input and the other is an output. The types of the two ports impose an extra restriction on whether they can be joined together. In the simplest case there is a set of possible types, and each can be joined only to itself. A more sophisticated idea is to define a subtype relation on the types, and to allow an output port of type t to be joined to an input of type u only if t is a subtype of u. I do this in chapter 4. Either option is compatible with any of the usual array of logical structures for the set of types.

Type systems of the form just described serve to ensure that only certain cuts can arise, and perhaps to ensure that rewrite rules exist for all of them. In addition, Lafont introduced what he called *semi-simple* nets, which obey a type system designed to prevent vicious circles. A vicious circle is a form of deadlock in which the principal path from a node leads back to that node. While this property is certainly desirable, the price is that to check the types of a program it must be constructed in a particular order; the locality that is one of Interaction Nets' most appealing features is lost. Researchers accustomed to term rewriting systems feel comfortable writing judgements about entire terms, and ideas imported from the lambda calculus often end up in this form [**7**]. I didn't do this.

My own type system must cope with linking. I was keen to ensure that programs can be linked even though they were written independently. This necessitated a denotational flavour: the type of a program makes a statement about its message-passing behaviour that can be understood independently of any identifiers defined within the program. I copied the idea from part of the programming language OCaml, but I think it is new to Interaction Nets.

2.1.6. Implementation. Jorge Sousa Pinto has done some work on implementing interaction nets [**29**] which has led to a useful tool for calculating the normal forms of programs. This in turn has been used for empirical studies of different encodings of the lambda calculus, for example [**21**, section 9].

As an exercise in programming, this work is not very ambitious. The paper does not say what programming language was used, but it looks like it was something like ML. It does not tell us much about the mapping of interaction nets onto real hardware. Instead, the principal interest of this paper is the data structure it uses to model an interaction net. This data structure is designed to minimise the cost of modifications *at the cuts*, since that is where rewriting happens.

To this end, it breaks a net down into what Lafont would call *principal nets*: singly-connected graphs in which every internal wire joins a principal port to an auxiliary port. The resulting forest is supplemented by information about the remaining wires, with a special case for the cuts. Mackie traces this idea back to Lafont's textual notation for writing interaction net rewrite rules, and has used it in his own textual notations for interaction nets [**10**], but Pinto can take credit for recognising its importance for an implementation. I followed the basic idea in the textual notation I invented for the proofs in chapter 3. Beyond this thesis, I plan to follow it again in my own implementation work. It is clearly the right approach.

The paper continues with a claim that the algorithm can be easily extended to exploit multiple processors. I do not believe this claim. It appears to be based on a memory model that has not been typical of any high-performance computer since the 1980s, in which grabbing a lock is barely more expensive than accessing memory. The algorithm grabs locks at least once per rewrite. The reality is that grabbing a lock forces the processor to flush much of its cache, which can be very expensive. A realistic implementation would not benefit from an extra processor unless it could perform at least tens of rewrites for each time it grabs a lock. Thus, Pinto's paper has not addressed the principal problem. Neither have I.

2.1.7. Encodings. A crucial ingredient in any project to use interaction nets as an intermediate language in a compiler is an array of encodings of other languages into interaction nets. This being a mainly theoretical problem, a considerable amount of work has been spent on it.

The lambda calculus and its relatives are especially favoured as source languages. In addition, it happens that Lamping's algorithm for optimal reduction of lambda calculus terms is an interaction net system [**1**], as are various of its later improvements. Mackie presents one of the best encodings of a lambda calculus variant (linear logic) in [**21**], which also includes a brief empirical review of the competition. One of the reasons that the lambda calculus is so favoured is that interaction nets are an excellent tool for studying sharing. Being linear themselves, there is no way that they can hide a copy or discard operation; all such operations must be encoded explicitly.

My interest in interaction nets, and in process calculi in general, is different: message passing is an excellent abstraction of IO, and of interaction between programs that are executing separately. On this level I feel it is a pity that interaction nets have mainly been applied to functional languages, whose model of IO and concurrency is invariably painful, and which do not provide the test cases for interaction nets to shine in a way that they could. I have tried to bring out this neglected side in the practical half of my thesis, especially chapter 6.

2.2. Process Calculi

From my perspective, the canonical process calculus is CCS [**23**]. Although other early process calculi such as Petri Nets [ref] and CSP [ref] have their descendants, almost all of the calculi I have encountered trace their lineage back to CCS, mostly via some variant of the π-calculus.

If I may deliberately over-simplify, CCS was thrust upon a world that thought of the lambda calculus as the core calculus that underlies all computation. Numerous other models existed, such as Turing machines, but they all fell in the same camp, which I think of as the lambda calculus camp. Programs were applied to inputs, ran for a while, and then returned results. When applied to the same inputs again, they produced the same results (ideally). Of course, there was also systems software, but that was written in low-level languages like assembler and C/Posix and was beyond all hope of comprehension.

That is, until CCS. Suddenly, here was a calculus that was designed to model concurrent, non-deterministic systems such as networks, user interfaces, web browsers and schedulers. What is more, it came with maths. In the same way that traditional (lambda calculus) programs could be optimised as long as the answers didn't change for any inputs, CCS provided criteria for validating optimisations of concurrent programs. That's the vision I'm chasing, twenty years later.

2.2.1. CCS. In this section, everything comes from [**23**] unless otherwise stated.

The philosophical message of CCS is that the interesting events are the ones that require the cooperation of two parties (*processes*).[2] Sometimes you hear people refer to this cooperation as message passing, and this has contributed words like 'input' and 'output' to the jargon. This is unfortunate, as communication in CCS is not message passing: it is *synchronisation*. The difference is that when two processes try to synchronise, both have to wait until the event happens, whereas for message passing only the receiver has to wait. The first true message-passing calculus in the CCS family was the asynchronous π-calculus (see section 2.3.3), some seven years later *[check]*.

2.2.1.1. *Syntax.* I expect most of my readers are familiar with CCS and so I will not dwell on its motivation and interpretation. However, I do depend on it quite heavily, and so I will give a complete definition. The syntax is that accepted by the following grammar, in which $\langle\text{name}\rangle$ is a countably infinite set of names:

DEFINITION 2.2.1. (CCS)

$$\begin{aligned}
\langle\text{action}\rangle &::= \overline{\langle\text{name}\rangle} \mid \langle\text{name}\rangle \mid \tau \\
\langle\text{sum}\rangle &::= 0 \mid \langle\text{sum}\rangle + \langle\text{sum}\rangle \mid \langle\text{action}\rangle \,.\, \langle\text{process}\rangle \\
\langle\text{process}\rangle &::= \langle\text{sum}\rangle \mid \langle\text{process}\rangle|\langle\text{process}\rangle \mid \nu\,\langle\text{name}\rangle \,.\, \langle\text{process}\rangle
\end{aligned}$$

In the process $\nu x.P$ the name x is bound in P. All other names are free. There are no side-conditions.

2.2.1.2. *Operational semantics.* The simplest way to define the semantics of CCS is to imagine processes swimming around in some sort of soup, and synchronising only when a sender happens by chance to encounter a matching receiver. This idea first appeared in [**3**]. Terms of the CCS grammar represent configurations of the soup, and reachability of configurations by swimming is represented by an equivalence relation called the structural congruence.

DEFINITION 2.2.2. (Structural congruence)

Let the *structural congruence*, written $\equiv$, be the smallest congruence satisfying the following axioms:

$$\frac{}{0 \mid P \equiv P} \qquad \frac{}{P \mid Q \equiv Q \mid P} \qquad \frac{}{P \mid (Q \mid R) \equiv (P \mid Q) \mid R}$$

$$\frac{}{0 + S \equiv S} \qquad \frac{}{S + T \equiv T + S} \qquad \frac{}{S + (T + U) \equiv (S + T) + U}$$

[2]Curiously, none of the calculi that use CCS-like synchronisation seem to address the question of more than two parties cooperating; it appears to be impossible to reduce three-way synchronisation cleanly to a series of two-way synchronisations without leaving some observable artifacts. This problem goes away in the asynchronous case (see section 2.3.3).

$$\frac{x \notin \text{fn}(Q)}{(\nu x.P) \mid Q \equiv \nu x.(P \mid Q)} \qquad \frac{}{\nu x.\nu y.P \equiv \nu y.\nu x.P}$$

The first line makes | a commutative monoid, and the second does the same for $+$. Finally, the third line gives some trivial properties of fresh names.

Note that I have been a little lazy here in not distinguishing sums from processes; although every sum is a process, its parse tree as a sum is not the same as its parse tree as a process. Arguably there ought to be a rule that if $S \equiv T$ (as sums) then $S \equiv T$ (as processes).

Synchronisation due to chance encounters can now be captured very simply.

DEFINITION 2.2.3. (Reaction relation)

Let the *reaction relation*, written $\longrightarrow$ be defined by the following rules:

$$\frac{}{\overline{x}.P + S \mid x.Q + T \longrightarrow P \mid Q} \qquad \frac{}{\tau.P + S \longrightarrow P}$$

$$\frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q} \qquad \frac{P \longrightarrow P'}{\nu x.P \longrightarrow \nu x.P'} \qquad \frac{P \equiv\longrightarrow\equiv P'}{P \longrightarrow P'}$$

This is a very pretty intuitive picture of the way computation happens in CCS. Its main drawback is that the derivation of a reaction may be exceedingly perverse, and usually won't respect the structure of a process. Indeed, a reaction can completely change the structure of a process, using the structural congruence. This makes it rather difficult to prove things about all reactions. This failing is corrected by a second definition (historically the first) of what is essentially the same behaviour.

DEFINITION 2.2.4. (Labelled transition relation)

Let the *labelled transition relation*, with elements written $P \xrightarrow{\alpha} P'$, P being a process or sum, P' being a process and α an action, be the smallest satisfying the following axioms:

$$\frac{}{\alpha.P \xrightarrow{\alpha} P}\text{PREFIX} \qquad \frac{S \xrightarrow{\alpha} P'}{S + T \xrightarrow{\alpha} P'}\text{SUM}_1 \qquad \frac{T \xrightarrow{\alpha} P'}{S + T \xrightarrow{\alpha} P'}\text{SUM}_2$$

$$\frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}\text{PAR}_1 \qquad \frac{P \xrightarrow{\overline{x}} P' \quad Q \xrightarrow{x} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}\text{COM}_1$$

$$\frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'}\text{PAR}_2 \qquad \frac{P \xrightarrow{x} P' \quad Q \xrightarrow{\overline{x}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}\text{COM}_2$$

$$\frac{P \xrightarrow{\alpha} P' \quad x \notin \text{fn}(\alpha)}{\nu x.P \xrightarrow{\alpha} \nu x.P'}\text{NEW}$$

Derivations of labelled transitions are completely guided by the structure of a process. In particular, the labelled transitions of $P \mid Q$ are completely determined by the labelled transitions of P and Q, irrespective of their internal structure. The price is that it is not at all obvious that | is a commutative monoid (for example).

The two definitions are related as follows:

THEOREM 2.2.5. *(Relationship of reactions and labelled transitions)*

$$\longrightarrow \; = \; \xrightarrow{\tau}\equiv$$

MIN too has a structural congruence, a reaction relation and a labelled transition relation, related by a theorem just like this.

2.2.1.3. *Infinite behaviour. [Cut down this section to a single paragraph explaining "not explained here"? It would mess up the introduction to the next section.]*

So far, I have only presented the finite subset of CCS. To obtain a computationally complete system, we need some sort of infinite behaviour. Conventionally, this is done using recursion. My favourite approach is to add μ binders and substitutions to the grammar. There is no need to present the details here.

2.2.2. Simulation. The content of this section is also mostly from [**23**].

The possibility of deriving all labelled transitions of $P \mid Q$ from the labelled transitions of P and of Q, and similarly for all the other CCS operators, along with the fact that these can be used to recover the reaction relation, strongly suggests that the entire behaviour of a process is contained in the labelled transition relation. This provides a way of defining when two processes are equivalent, namely, when they have the same labelled transitions.

That the labelled transitions *contain* the entire behaviour of a process is quite a weak statement. After all, the detailed syntax also contains the entire behaviour. We can argue that a definition of equivalence will be sound (that is, if two processes satisfy the definition then they are truly equivalent) but so far we have no way of arguing that it is complete (any two processes that are truly equivalent satisfy the definition). We will fix this (and define 'truly equivalent') in section 2.4.1.

In section 2.2.1.1, I gave several examples of processes that I said were equivalent because after some small number of synchronisations they became identical. For example, $\overline{x}.P \mid x.Q$ and $\overline{x}.(P \mid x.Q)+x.(\overline{x}.P \mid Q)+\tau.(P \mid Q)$ are related in this way, as are $\nu x.(x.P + y.Q)$ and $y.\nu x.Q$. However, we must be bolder than that. There are plenty of examples of pairs of programs which take an arbitrarily large number of synchronisations to become equal, and plenty more which are equivalent even though they have no common future.

The technology required to handle the infinite cases is simulation. Again I will cover this material thoroughly, despite expecting many of my readers to be familiar with it, because I depend on it heavily.

2.2.2.1. *Strong simulation.*

DEFINITION 2.2.6. (Strong simulation)

Say a relation $\underline{S}$ on processes is a *strong simulation* iff $\underline{S} \xrightarrow{\alpha} \subseteq \xrightarrow{\alpha} \underline{S}$; that is, iff it satisfies the following:

- For all processes P and Q and actions α, if $P \ \underline{S} \ Q$ and $Q \xrightarrow{\alpha} Q'$ then there exists a process P' such that $P \xrightarrow{\alpha} P'$ and $P' \ \underline{S} \ Q'$.

Say P *strongly simulates* Q iff there exists a strong simulation $\underline{S}$ such that $P \ \underline{S} \ Q$. Say P and Q are *mutually strongly similar* iff P simulates Q and Q simulates P.

Say a simulation $\underline{S}$ is a *strong bisimulation* iff it is symmetric. Say P and Q are *strongly bisimilar*, and write $P \sim Q$, iff there exists a strong bisimulation $\underline{R}$ such that $P \ \underline{S} \ Q$.

This definition is quite subtle. Nowhere does it mention that a simulation must be transitive or reflexive. However, $=$ is a simulation, and the composition of two simulations is a simulation, so any simulation can be extended to be reflexive and transitive. Also, $=$ is a bisimulation, and the composition of two bisimulations is a bisimulation, so bisimulations can be similarly extended. The definition also does not require that the 'simulates' relation is a simulation, but in fact it is the largest simulation, and similarly bisimilarity is the largest bisimulation. Therefore, mutual similarity and bisimilarity, in addition to being symmetric by definition, are reflexive and transitive; that is, they are equivalence relations.

One of the most important strong bisimulations is the structural congruence. Proving that the structural congruence is a strong bisimulation is the hard part of proving theorem 2.2.5.

I have defined two equivalence relations. Which is correct? Both turn out to be congruences for CCS. Mutual similarity contains bisimilarity, because any bisimulation is both a simulation and the inverse of a simulation. The inclusion is strict because $a.b.0 + a.0$ mutually simulates $a.b.0$ but is not bisimilar to it. The difference between these two processes is that the former can get stuck after synchronising on a, whereas the latter will always go on to try to synchronise on b.

Each equivalence relation is appropriate in some applications. In applications where all possible paths are explored, such as an abstract interpretation, mutual similarity may be more appropriate. In applications where one possible path is chosen, such as a program, bisimilarity may be more appropriate. Still other equivalences may be appropriate in other situations. Van Glabeek has written an excellent review of 12 of the more commonly encountered strong equivalences [**12**].

2.2.2.2. *Weak simulation.* Strong bisimilarity only relates programs that perform exactly the same number of silent transitions. For example, P is not (in general) strongly bisimilar to $\tau.P$, even though the only thing the latter can do is to become the former. It many applications, including mine, this is no good, as reducing the number of computation steps is one of the goals. In such situations, weak similarity may be more appropriate.

DEFINITION 2.2.7. (Weak transition relation)

Let $\overset{\tau}{\Longrightarrow}$ mean $\xrightarrow{\tau}^{*}$ and let $\overset{\alpha}{\Longrightarrow}$ mean $\Longrightarrow\xrightarrow{\alpha}\Longrightarrow$ if $\alpha \neq \tau$.

Define *weak simulation*, *weak simulates*, *weak mutual similarity* (written $\lessapprox$), *weak bisimulation* and *weak bisimilarity* (written $\approx$) just like their strong counterparts, but using $\overset{\alpha}{\Longrightarrow}$ everywhere in place of $\xrightarrow{\alpha}$.

A practical failing of this definition is that $\overset{\alpha}{\Longrightarrow}$ is not an image-finite relation: for some P and α there may be infinitely many processes P' such that $P \overset{\alpha}{\Longrightarrow} P'$. This is easily fixed, by using the following alternative definition of weak simulation:

DEFINITION 2.2.8. (Weak simulation)

Say a relation $\underline{S}$ on processes is a *weak simulation* iff $\underline{S} \xrightarrow{\tau} \subseteq \overset{\tau}{\Longrightarrow} \underline{S}$ and $\underline{S} \xrightarrow{\alpha} \subseteq \overset{\alpha}{\Longrightarrow} \underline{S}$.

Weak bisimilarity is a much larger relation than strong bisimilarity, and opens up new possibilities. For example, the weak bisimilarity $\mu V.(S + \tau.(T + \tau.V)) \approx S + T$ can be used to reduce all sums to the form $\alpha.P + \tau.Q$. This 'busy wait' construction has found numerous applications [**11, 25**].

2.2.2.3. *'Up to' techniques.* The material in this section comes from a variety of sources, including [**23**], but is most conveniently collected in [**33**, section 2.3].

Constructing simulations can be a chore. To prove two processes equivalent can require an infinite relation if either process has infinite behaviour (which is the most interesting case), but proving properties of infinite sets is rarely easy. To reduce the work, it becomes essential to develop techniques that capture recurrent parts of these proofs.

DEFINITION 2.2.9. (Strong simulation precursor)

Given a relation $\underline{S}$ define $\hat{\underline{S}}$ to be the smallest reflexive, transitive relation containing $\underline{S}$ and $\sim$ (and hence $\equiv$) and satisfying the following rules:

$$\frac{P \;\hat{\underline{S}}\; Q}{\alpha.P+T \;\hat{\underline{S}}\; \alpha.Q+T} \qquad \frac{P_1 \;\hat{\underline{S}}\; Q_1 \quad P_2 \;\hat{\underline{S}}\; Q_2}{P_1 \mid P_2 \;\hat{\underline{S}}\; Q_1 \mid Q_2} \qquad \frac{P \;\hat{\underline{S}}\; Q}{\nu x.P \;\hat{\underline{S}}\; \nu x.Q} \qquad \frac{P \;\hat{\underline{S}}\; Q}{P\sigma \;\hat{\underline{S}}\; Q\sigma}$$

Say $\underline{S}$ is a *strong simulation precursor* iff $\underline{S} \xrightarrow{\alpha} \subseteq \xrightarrow{\alpha} \hat{\underline{S}}$.

THEOREM 2.2.10. *(Strong simulation up to various things)*

If $\underline{S}$ is a strong simulation precursor, then $\hat{\underline{S}}$ is a strong simulation.

Note that any simulation is a simulation precursor, so it is certainly no harder to find a simulation precursor than to find a simulation. In other words, there's never a reason not to use the technique. Also, any simulation found without this technique can automatically be enlarged; this is a good way of showing that mutual similarity and bisimilarity are congruences, for example.

A good illustration of the power of the technique is to attempt to prove the strong bisimilarity of $P = \mu V.x.V$ and $Q = \mu W.(x.P + x.(W \mid W))$, both of which just perform xs for ever. The behaviour of P is very simple: $P \xrightarrow{x} P$. However, a typical reduction sequence of Q is $Q \xrightarrow{x}{}^{7} Q \mid (((P \mid Q) \mid (Q \mid Q)) \mid P)$. To construct a raw bisimulation, we would have to find a way to enumerate all such processes reachable from Q, and to relate each of them to and from P. Then we'd have to prove that the relation really is a bisimulation. In contrast, it is sufficient to observe that the symmetric closure of $\{(P,Q),\ (Q, Q \mid Q)\}$ is a simulation precursor.

A similar technique, but with $\approx$ in place of $\sim$ and $\xRightarrow{\alpha}$ in place of $\xrightarrow{\alpha}$, can be used for constructing weak simulations. However, it is nice to keep the image-finite $\xrightarrow{\alpha}$ in the premise of the definition. We must sacrifice some power in order to do this. The most painful loss is transitivity, but we also partly lose the ability to work up to $\approx$.

DEFINITION 2.2.11. (Weak simulation precursor)

Given a relation $\underline{S}$ on processes, define $\hat{\underline{S}}$ to be the smallest relation containing $\underline{S}$ and $\approx$ and satisfying the following rules:

$$\frac{P_1 \;\hat{\underline{S}}\; Q_1 \quad \ldots \quad P_n \;\hat{\underline{S}}\; Q_n}{\alpha_1.P_1 + \ldots + \alpha_n.P_n \;\hat{\underline{S}}\; \alpha_1.Q_1 + \ldots + \alpha_n.Q_n}$$

$$\frac{P_1 \;\hat{\underline{S}}\; Q_1 \quad P_2 \;\hat{\underline{S}}\; Q_2}{P_1 \mid P_2 \;\hat{\underline{S}}\; Q_1 \mid Q_2} \qquad \frac{P \;\hat{\underline{S}}\; Q}{\nu x.P \;\hat{\underline{S}}\; \nu x.Q} \qquad \frac{P \;\hat{\underline{S}}\; Q}{P\sigma \;\hat{\underline{S}}\; Q\sigma}$$

Say $\underline{S}$ is a *weak simulation precursor* iff $\underline{S} \xrightarrow{\tau} \;\subseteq\; \xRightarrow{\tau} \approx \hat{\underline{S}} \sim$ and $\underline{S} \xrightarrow{\alpha} \;\subseteq\; \xRightarrow{\alpha} \approx \hat{\underline{S}} \approx$.

THEOREM 2.2.12. *(Weak simulation precursor)*

If $\underline{S}$ is a weak simulation precursor then $\hat{\underline{S}}$ is a weak simulation.

This is enough to show that weak mutual similarity and weak bisimilarity are congruences,[3] which in turn is more than enough to obviate the following, otherwise

[3]The version of CCS given in **[23]** makes no distinction between sums and processes, so that contexts of the form $[\,]{+}P$ are syntactically correct (my version permits only $\alpha.[\,]{+}S$). Weak bisimilarity is then not a congruence, because $x.0 \approx \tau.x.0$ but $x.0 + y.0 \not\approx \tau.x.0 + y.0$ (only the right-hand process can internally commit not to synchronise on y). The fix I have adopted is widely known and used.

tempting addition to definition 2.2.11:

$$\frac{P \ \sim \hat{\underline{S}} \sim \ Q}{P \ \hat{\underline{S}} \ Q}$$

These are the only 'up to' techniques that I need in this thesis, but there are many more.

2.2.2.4. *Coupled similarity.* Weak bisimilarity is not big enough for my application, on account of one example: in general, $\tau.P+\tau.(\tau.Q+\tau.R) \not\approx \tau.(\tau.P+\tau.Q)+\tau.R$ although these are intuitively both just ways of choosing at random between P, Q and R. This problem was first fixed by Parrow and Sjödin [**26**], but a later work by Nestmann and Pierce [**25**] gives a more general solution.

The trick is to weaken the requirement on bisimulations that they be symmetric.

DEFINITION 2.2.13. (Coupled simulation)

Say a weak simulation $\underline{S}$ is *coupled* iff $\underline{S} \subseteq \overset{\tau}{\Longrightarrow} \underline{S}^{-1}$.

Say P *coupled simulates* Q iff there exists a coupled simulation $\underline{S}$ such that $P \ \underline{S} \ Q$. Say P and Q are *mutually coupled similar*, and write $P \eqsim_C Q$, iff P coupled simulates Q and Q coupled simulates P.

Like weak bisimilarity and mutual weak similarity, mutual coupled similarity is a congruence. However, unlike the 'weak simulates' relation, the 'coupled simulates' relation is not a precongruence. This is because $\overset{\tau}{\Longrightarrow}$ is not closed with respect to guarded contexts.

Arguably I should be looking for a still larger relation than mutual coupled similarity. I find it difficult to motivate the coupling condition intuitively. Certainly something is necessary, because mutual weak similarity is too large, but surely not something as strong as that? Having said that, mutual coupled similarity has a pretty definition that is easy to work with, and I have not yet encountered any examples that require anything larger.

2.3. Mobility

A failing of CCS is that the set of channels that two processes have in common, and which are therefore useful for synchronising, can never increase. This remains true despite the fact that either of the processes may break itself into lots of fragments; the original resources (at most) must be spread ever more thinly. The π-calculus was designed to fix this problem. The feature it adds, the ability to build new channels joining existing processes, is called *mobility*.

The π-calculus was first presented by Milner, Parrow and Walker in 1989, but since its publication in 2001 Sangiorgi and Walker's book "The π-calculus" [**33**] is the definitive resource. It really is a brilliant piece of work, which deserves to be preferred to the primary sources.

The fate of the π-calculus has been not so much widespread adoption as enormous influence. Very few authors have found it suits their purpose as it stands, but everybody (including me!) seems to want to invent a calculus just like it. It has therefore gradually acquired a role as the central repository of all maths on the subject of 'that sort of calculus'.

Figure 2.3.1 shows an approximate genealogy of MIN. Don't take it too seriously. It is intended to help you navigate the next few sections, in which I present the π-calculus and those of its many sub-calculi that have contributed ideas to the

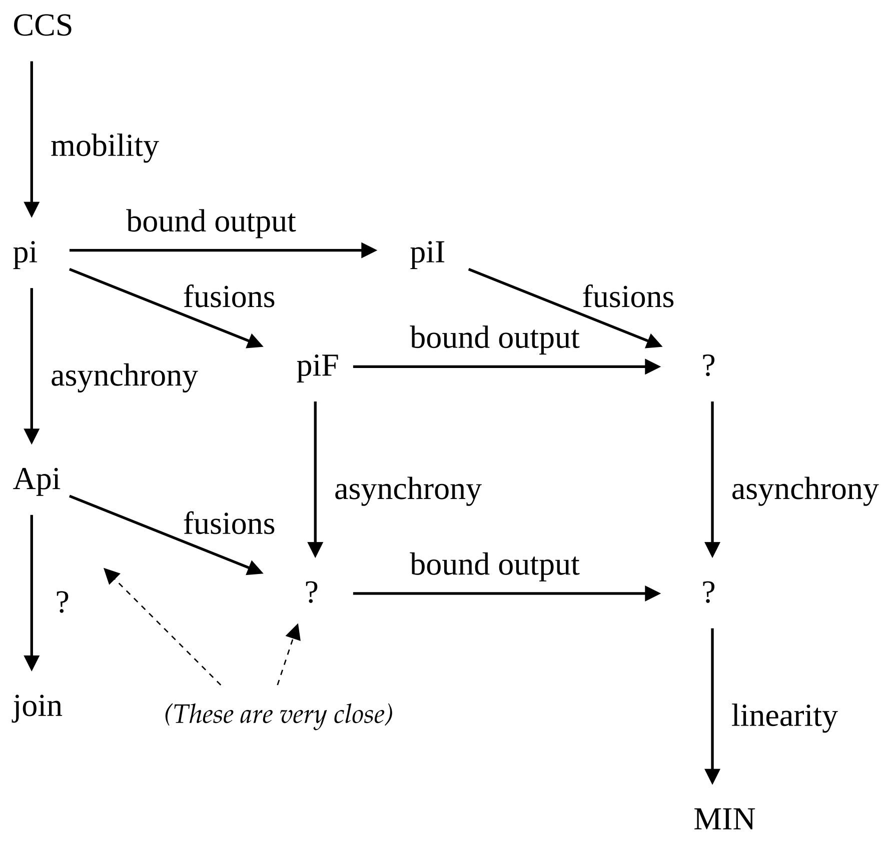

FIGURE 2.3.1. A MIN-centric genealogy of process calculi. Question marks indicate calculi or concepts which are sensible but for which there is no recognised brand name in the literature.

design of MIN. I review most but not all of the concepts in the figure. Fusions don't get a whole section, because they are important mainly for calculi that are not asynchronous, and MIN is. Linearity doesn't get a section because it is not background material; MIN is the first linear process calculus that I know of.

2.3.1. The π-calculus. The principal innovation in the π-calculus (relative to CCS) is in the guards. As well as synchronising on a channel, a sender and receiver negotiate an additional channel that can be used in subsequent communications. The negotiation is of a very simple form: the sender chooses a channel using any algorithm it pleases, and informs the receiver of its choice. The symmetry between the sender and receiver in CCS is therefore broken in the π-calculus.

There are two other innovations. First, *replication* is used in place of recursion as the mechanism for infinite behaviour. This simplifies the definition of the calculus, but does not change the expressive power. We can ignore replication here. Second, the π-calculus adds a *match* primitive, that allows a process to determine whether two channels are the same without synchronising on them. This does change the expressive power, in a way that is useful but not very well understood. I considered that grounds enough to omit it from MIN (it's not my battle) and I'll omit it here too.

2.3.1.1. *Syntax.* The new syntax for sending is $\overline{x}y.P$, in which x is the channel on which the process synchronises, y is the channel it has chosen, and P is the continuation process that will be activated if the synchronisation succeeds. y can be any channel; the idiom $\nu y.\overline{x}y.P$ ensures it is fresh, but that is not always what is desired. The new syntax for receiving is $x(z).Q$, in which x is the channel on which the process synchronises, and Q is the continuation process. The name z is a formal argument, that will be replaced everywhere in Q (except under νz or another input guard $w(z)$) by the sender's channel if the synchronisation succeeds. Therefore $x(z).Q$ and $x(v).Q[z \mapsto v]$ mean the same thing if Q does not contain v; this is alpha-conversion again.

In summary, the syntax of the finite subset of the π-calculus is that accepted by the following grammar:

DEFINITION 2.3.1. (π-calculus)

$$\begin{aligned}
\langle\text{prefix}\rangle &::= \overline{\langle\text{name}\rangle}\,\langle\text{name}\rangle \mid \langle\text{name}\rangle\,(\langle\text{name}\rangle) \mid \tau \\
\langle\text{sum}\rangle &::= 0 \mid \langle\text{sum}\rangle + \langle\text{sum}\rangle \mid \langle\text{prefix}\rangle\,.\,\langle\text{process}\rangle \\
\langle\text{process}\rangle &::= \langle\text{sum}\rangle \mid \langle\text{process}\rangle | \langle\text{process}\rangle \mid \nu\,\langle\text{name}\rangle\,.\,\langle\text{process}\rangle
\end{aligned}$$

The name x is bound in P in the processes $\nu x.P$ and $y(x).P$. All other names are free. There are no side-conditions.

2.3.1.2. *π-calculus semantics.* The reaction semantics of the π-calculus is very close to that of CCS. The structural congruence is completely unchanged, but we have to account for the new prefixes in the definition of the reaction relation.

DEFINITION 2.3.2. (Reaction relation)

Let the *reaction relation*, written $\longrightarrow$, follow the definition used for CCS but with the first rule changed to the following:

$$\frac{}{\overline{x}y.P + S \mid x(z).Q \longrightarrow P \mid Q[z \mapsto y]}$$

Though the reaction semantics is uncontroversial, the labelled transition semantics has been the subject of much debate, and a certain amount of confusion. There are several superficially plausible extrapolations from CCS. Should labels look like prefixes, so that $x(y).P \xrightarrow{x(y)} P$, or should they incorporate the substitution, so that $x(y).P \xrightarrow{xz} P[y \mapsto z]$? Plugging these two options into the standard definition of strong bisimilarity gives *ground* and *early* bisimilarity respectively. Another slightly different approach gives *late* bisimilarity.

Not only are these three equivalences all different, none of them is a congruence with respect to input prefix. The problem is that input prefixes can cause the same channel to be substituted for several names. This makes new connections in the program that the bisimulations have not taken into account. To resolve this, it is necessary to ensure that the relations are closed under substitutions. The largest substitution-closed relation contained in each of the three equivalences is finally a congruence. However, these congruences do not all coincide, either.

It is also possible to put the substitution-closure requirement into the definition of bisimulation, so that a different substitution can be chosen at every execution step. This leads to *open* bisimulation, yet another different congruence. Open bisimulation has arguably the prettiest definition, but it is not the largest congruence.

This monstrous mathematical miasma was most aggressively attacked by Sangiorgi. In [ref 1995?] he defines ground, late and early bisimulation uniformly in terms of a single labelled transition relation, the better to be able to compare them. Open bisimulation appears for the first time in the same paper. Writing later with Walker [**33**] early equivalence and congruence seem to be the favourite, but the book still includes a substantial discussion of the alternatives. If you are unfortunate enough to need to know the details, look there.

Note that this discussion has been about strong bisimilarity only. The other observational equivalences have the same problems all over again.

2.3.2. The πI-calculus. Another approach to the problem of defining bisimulation for the π-calculus is also due to Sangiorgi [**32**]. He observed that a slight redesign of the calculus removes essentially all of the complications, but barely changes the expressive power. The result is the πI-calculus. It is presented in section 5.7 of [**33**], where it is called the $P\pi$-calculus.

In my opinion, this is the right approach. When a design feature turns out to cause significant complications, it should in future be avoided. The πI-calculus offers a way of avoiding a particularly poor feature of the π-calculus. I therefore find it surprising that despite being widely known it seems to have been ignored. I have incorporated its central idea into MIN, and I urge any of my readers who find themselves designing a channel-passing calculus to do the same.

2.3.2.1. *Syntax.* What is this central idea, then? It is that outputs as well as inputs should be binding. The πI-calculus syntax for sending on a channel is $\overline{x}(y).P$. As usual, x is the channel on which to synchronise, and P is the continuation process, but the name y is a formal argument that will be replaced by a fresh channel everywhere in P (except under νy or $\overline{w}(y)$ or $w(y)$) when the synchronisation succeeds. In other words, it is a perfect symmetric counterpart of the input prefix $x(y).P$ which, now that only fresh channels can be sent, can only receive fresh channels.

In summary, the syntax of the finite subset of the πI-calculus is that accepted by the following grammar:

DEFINITION 2.3.3. (πI-calculus)

$$\begin{array}{rcl}\langle\text{action}\rangle & ::= & \overline{\langle\text{name}\rangle}(\langle\text{name}\rangle) \mid \langle\text{name}\rangle\,(\langle\text{name}\rangle) \mid \tau \\ \langle\text{sum}\rangle & ::= & 0 \mid \langle\text{sum}\rangle + \langle\text{sum}\rangle \mid \langle\text{action}\rangle\,.\,\langle\text{process}\rangle \\ \langle\text{process}\rangle & ::= & \langle\text{sum}\rangle \mid \langle\text{process}\rangle|\langle\text{process}\rangle \mid \nu\,\langle\text{name}\rangle\,.\,\langle\text{process}\rangle\end{array}$$

The name y is bound in $\overline{x}(y).P$ and $x(y).P$ and $\nu y.P$. All other names are free. There are no side-conditions.

2.3.2.2. *Operational semantics.* The structural congruence and reaction relation are much the same as for the π-calculus. The crucial rule is

$$\frac{}{\overline{x}(y).P + S \mid x(y).Q + T \longrightarrow \nu y.(P \mid Q)}$$

Note that there is no substitution in this rule; it is all absorbed into alpha-conversion. Note also that it is completely symmetric, just like CCS. In this way, the πI-calculus is a more faithful extrapolation from CCS than the π-calculus. This theme continues in the definition of the labelled transition relation.

DEFINITION 2.3.4. (Labelled transition relation)

Let the labelled transition relation, with elements written $P \xrightarrow{\alpha} P'$, P being a sum or process, P' being a process, and α being an action, be the smallest relation satisfying the following rules:

$$\frac{}{\alpha.P \xrightarrow{\alpha} P}\text{PREFIX} \qquad \frac{S \xrightarrow{\alpha} P'}{S+T \xrightarrow{\alpha} P'}\text{SUM}_1 \qquad \frac{T \xrightarrow{\alpha} P'}{S+T \xrightarrow{\alpha} P'}\text{SUM}_2$$

$$\frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}\text{PAR}_1 \qquad \frac{P \xrightarrow{\overline{x}(y)} P' \quad Q \xrightarrow{x(y)} Q'}{P \mid Q \xrightarrow{\tau} \nu y.(P' \mid Q')}\text{COM}_1$$

$$\frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'}\text{PAR}_2 \qquad \frac{P \xrightarrow{x(y)} P' \quad Q \xrightarrow{\overline{x}(y)} Q'}{P \mid Q \xrightarrow{\tau} \nu y.(P' \mid Q')}\text{COM}_2$$

$$\frac{P \xrightarrow{\alpha} P' \quad x \notin \mathrm{n}(\alpha)}{\nu x.P \xrightarrow{\alpha} \nu x.P'}\text{NEW}$$

Note that, in the derivation of a synchronisation, it is the COM rules that actually choose the fresh name; this is represented by the appearance of νy in their conclusions.

Strong and weak simulation can be defined uncontroversially just like CCS, and all the various equivalences turn out to be congruences naturally, without messing around with substitutions. There is no tension between the various definitions; they are all simultaneously just like CCS. This is reflected in the fact that there is only one plausible definition for each of the various observational equivalences. The problem is solved.

Although bound output can be achieved in the ordinary π-calculus using the idiom $\nu y.\overline{x}y.P$, the impossibility of free output in the πI-calculus makes the two calculi quite different. Processes that are not equivalent in the π-calculus can be equivalent in the πI-calculus, even if they make no use of free output, because they will never be subjected to substitutions. In fact, the unique strong bisimilarity relation in the πI-calculus is strictly larger than all of the π-calculus strong bisimilarity congruences.

2.3.3. Asynchronous π-calculus. The asynchronous π-calculus is the best known sub-calculus of the π-calculus, and has many sub-calculi of its own. It was invented by Honda. His original paper [**13**] contains a conjecture that turned out to be incorrect, so I recommend finding a later source, such as [**25**] or sections 5.1 to 5.5 of [**33**].

Like the πI-calculus, the asynchronous π-calculus can be seen as a syntactic restriction of the π-calculus, and like the πI-calculus, this viewpoint does not do it justice at all. It is motivated by the many applications in which the primitive communication medium does not support synchronisation, as found in CCS and the π-calculus, but instead provides a reliable way of sending messages.

Message-passing media are the rule, not the exception. Unix pipes and signals, and the various internet protocols, are all based on message passing. A subroutine call is a sequence of two messages (or just one if the subroutine doesn't return). There are also examples, such as radio and human speech, in which a message will be lost if the receiver is not listening; in practice these are always wrapped in buffers and re-transmission protocols in order to make a reliable communication channel, and so they too are best regarded as message-passing media.

Asynchronous communication also has significant advantages over synchronisation. It is possible to write an asynchronous π-calculus process that forwards messages from one channel to another, in such a way that a sender and a receiver connected by a chain of forwarders can communicate as if the forwarders weren't there. It is also possible to encode multi-way communication, with one receiver but several senders, into the two-way communication that the calculus provides as primitive, without any observable artifacts. The impossibility of these feats in ordinary π-calculus suggests that maybe it is not all there [ref πF-calculus].

2.3.3.1. *Syntax.* The syntactic restriction that turns the π-calculus into the asynchronous π-calculus is to forbid output guards from guarding anything. The syntax for sending a message is simply $\overline{x}y$. The name x identifies the channel on which to send, and the message is the name y. There is no continuation process. Also, messages are processes, not sums, and so the $+$ operator can't be applied to them.

The syntax of the finite subset of the asynchronous π-calculus is that accepted by the following grammar:

DEFINITION 2.3.5. (Asynchronous π-calculus)

$$\begin{array}{rcl} \langle\text{sum}\rangle & ::= & 0 \mid \langle\text{sum}\rangle + \langle\text{sum}\rangle \mid \langle\text{name}\rangle\,(\langle\text{name}\rangle).\,\langle\text{process}\rangle \\ \langle\text{process}\rangle & ::= & \langle\text{sum}\rangle \mid \overline{\langle\text{name}\rangle}\,\langle\text{name}\rangle \mid \langle\text{process}\rangle|\langle\text{process}\rangle \mid \nu\,\langle\text{name}\rangle\,.\,\langle\text{process}\rangle \end{array}$$

The name y is bound in $x(y).P$ and $\nu y.P$. All other names are free. There are no side-conditions.

2.3.3.2. *Operational semantics.* The definition of the structural congruence and the reaction relation closely follows the π-calculus. The crucial rule is this:

$$\frac{}{\overline{x}y \mid x(z).P + S \longrightarrow P[z \mapsto y]}$$

It seems quite difficult to come up with a tidy definition of the labelled transition relation, by which I mean one in which the derivation of a transition is structured according to the syntax of the process. Most attempts in the literature either appeal to the structural congruence [**14**] or modify the definition of bisimilarity [**25, 33**]. I have found a proper structure-respecting definition for MIN, but it does not adapt well to the asynchronous π-calculus. I will therefore just follow [**14**].

DEFINITION 2.3.6. (Labelled transition relation)

Let the *actions* be defined by the following grammar:

$$\langle\text{action}\rangle \quad ::= \quad \tau \mid xy \mid \overline{x}y \mid \overline{x}(y)$$

Let $\text{n}(\alpha)$ be the set of all names in an action α, and let $\text{bn}(\alpha)$ be the subset of names inside brackets.

Let the *labelled transition relation* be the smallest ternary relation satisfying the following rules:

$$\frac{}{0 \xrightarrow{xy} \overline{x}y} \qquad \frac{}{\overline{x}y \xrightarrow{\overline{x}y} 0} \qquad \frac{}{\overline{x}y \mid x(z).P \xrightarrow{\tau} P[z \mapsto y]}$$

$$\frac{P \xrightarrow{\alpha} P' \quad \text{bn}(\alpha) \cap \text{fn}(Q) = \{\}}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad \frac{P \xrightarrow{\alpha} P' \quad z \notin \text{n}(\alpha)}{\nu z.P \xrightarrow{\alpha} \nu z.P'} \qquad \frac{P \xrightarrow{\overline{x}y} P' \quad x \neq y}{\nu y.P \xrightarrow{\overline{x}(y)} \nu y.P'}$$

$$\frac{P \equiv\xrightarrow{\alpha}\equiv P'}{P \xrightarrow{\alpha} P'}$$

Note that none of the input or output actions are used to derive τ steps. In particular, the motivation behind the distinction between the two kinds of output action $\overline{x}y$ and $\overline{x}(y)$ is far from clear. Superficially, there is no reason to believe that bisimilarity over these transitions will be a congruence. Surprisingly, however, ground, early and late bisimilarity all coincide, and moreover are closed under substitutions, so they are congruences and coincide with open bisimilarity. This is true of both the strong and weak cases.

The congruences are rather larger than any of the corresponding π-calculus congruences. For example, $x(y).\overline{x}y$ is weak bisimilar to 0, despite the fact that the former reads from the channel x. This reflects the fact that no asynchronous π-calculus process can observe that a message it has sent has been received. When the asynchronous π-calculus first appeared, many people dismissed it as trivial because they missed this important point.

2.3.4. Join calculus. The join calculus is an asynchronous process calculus, formed by imposing a further syntactic restriction on the asynchronous π-calculus [**11**]. Although I discovered it long after fixing the design of MIN, it deserves a mention here because it shares motivation and therefore design decisions with MIN. It gets rather little attention from [**33**], so don't look there.

The syntactic restriction is a discipline on the use of restriction and summation. Both are replaced by a single syntactic construction. Approximately, the join calculus term def $x(z).P + y(z).Q$ in R has the same operational semantics as the asynchronous π-calculus term $\nu x.\nu y.((x(z).P + y(z).Q) \mid R)$. However, this is not the conventional syntax, and does not do justice to the richer construct that is in fact available. I refer you to [**11**] for the full details.

This construction shares with MIN a determination to control the scope of sums, in some way, to facilitate linking and to permit an efficient implementation. In the join calculus, this is achieved by controlling the scope of the channels on which the sums are listening. In MIN it is achieved by naming the sums (which I call destructors), and restricting the scope of their names. I think this difference follows from my decision to make MIN a linear calculus, but I'm not sure. Certainly, communication in the join calculus is many-to-one, but in MIN is one-to-one (in the asynchronous π-calculus it is many-to-many).

An interesting consequential difference is that the scope of a join calculus sum can be extruded just like any other name; in MIN destructors are not communicable values, so the effect must be achieved using a different mechanism: wires.

2.4. Calculus-independent Technology

This section describes two important ideas about process calculi that I have bought.

2.4.1. Reaction-based process semantics. The distinction between asynchronous bisimilarity and the restriction of (any) π-calculus bisimilarity to the asynchronous π-calculus was the catalyst for interest in reaction-based equivalences. Although there was originally no real argument to support the claim, everybody accepted that bisimilarity for CCS was about as large as it could be. An argument finally appeared in [**24**], for the π-calculus. However, the asynchronous π-calculus provided the first clear example in which the obvious approach (the one extrapolated from CCS) gave a relation that was much smaller than it should be.

In order to drive this point home, Honda and Yoshida set out to construct the largest bisimulation-like equivalence for the asynchronous π-calculus that they could [**14**]. The details of their construction are a little esoteric, and I will gloss over them enough not to encounter differences from the approach in [**24**]. The important point is that the definition must depend on the syntax of the calculus, the reaction relation, and as little else as possible.

In particular, the definition must not depend on first choosing the right labelled transition relation. On the contrary, it should provide a way of arguing that the labelled transition relation is right. In my opinion, a result of this sort is an important part of the theory of any process calculus.

An attractive feature of the approach is that it is almost entirely portable between calculi. Using the terminology of [**14**], an observational equivalence relation should satisfy three properties: it should be *context-closed*, *reduction-closed* and *sound*.

The definition of context-closure is uncontroversial:

DEFINITION 2.4.1. (Context-closed)

Say a relation $\underline{R}$ is *context-closed* iff $P \ \underline{R}\ Q$ implies $C[P]\ \underline{R}\ C[Q]$ for all processes P and Q and syntactic contexts $C[\,]$.

The definition of reduction-closure can be varied to obtain different kinds of bisimilarity. This version looks like, and leads to, weak bisimilarity:

DEFINITION 2.4.2. (Reduction-closure)

Say a relation $\underline{R}$ is *reduction-closed* iff it is symmetric and $\underline{R} \longrightarrow\ \subseteq\ \longrightarrow^* \underline{R}$.

The purpose of the soundness condition is to ensure that not all processes are related. The definition can vary greatly, but the general form is that a relation $\underline{R}$ is sound if $P\ \underline{R}\ Q$ implies $f(P) = f(Q)$ for some function f that must be chosen in an ad-hoc way for a particular calculus and application. The freedom of choice available here is uncomfortable, but it is not a problem in practice, as almost any calculus comes with some property that is obviously enough to distinguish processes. For example, this definition is appropriate for the asynchronous π-calculus:

DEFINITION 2.4.3. (Soundness)

Define the predicate $f(P)$ on processes P to be true iff $P \longrightarrow^* \equiv \overline{x}y \mid P'$ for some names x and y and process P'.

Say a relation $\underline{R}$ on processes is *sound* iff $f(P) = f(Q)$ whenever $P\ \underline{R}\ Q$.

The most important property of soundness is that if $\underline{R}$ is sound then so is $\underline{R}\,\underline{R}$. If this property does not hold, then there is no guarantee that the largest sound relation of a particular kind is transitive. One cannot explicitly ask for the largest transitive sound relation of a particular kind, because the axiom for transitivity involves as premises two elements of the relation: if not $P_1\ \underline{R}\ P_3$ then which of the elements $P_1\ \underline{R}\ P_2$ and $P_2\ \underline{R}\ P_3$ should be removed? This mistake is subtle and one sometimes sees it in the literature [**11**]. The form given above, in terms of a function, is guaranteed to have the required property.

We are now ready to define the observational equivalence:

DEFINITION 2.4.4. (Barbed bisimilarity)

Say two processes P and Q are *barbed bisimilar* iff $P\ \underline{R}\ Q$ for some context-closed, reduction-closed, sound relation $\underline{R}$.

Sometimes one sees the following definition instead:

DEFINITION 2.4.5. (Barbed congruence)

Say two processes P and Q are *barbed congruent* iff $C[P] \ \underline{R}\ C[Q]$ for all contexts $C[\,]$ for some reduction-closed, sound relation $\underline{R}$.

In other words, context-closure is required only at the very first step of the bisimulation game. This makes no difference for the asynchronous π-calculus. For MIN I have used the first form.

As an unintended side-effect of establishing that the ordinary π-calculus congruences are too small for the asynchronous π-calculus, Honda and Yoshida managed to show that their own, larger, asynchronous bisimilarity was also too small. The awkward example involves an *equator* process:

$$E_{xy} = \mu V.(x(z).(\overline{y}z \mid V) + y(z).(\overline{x}z \mid V))$$

Intuitively, E_{xy} simply establishes a perfect connection between the names x and y, so $E_{xy} \mid P[y \mapsto x]$ and $E_{xy} \mid P[x \mapsto y]$ should be (weakly) equivalent for all P.[4] They are indeed weak barbed bisimilar. However, $E_{xy} \mid \overline{z}x$ and $E_{xy} \mid \overline{z}y$ are not weak labelled bisimilar.

I am not aware of any work that successfully bridges the gap between the labelled and barbed bisimilarities for the asynchronous π-calculus. In MIN, thanks to the technology of the πI-calculus, labelled and barbed bisimilarity do coincide.

Here is a table summarising whether or not some important calculi have labelled congruences that match their barbed congruences:

Calculus	Labels okay?
CCS	yes
π	no
πI	yes
$A\pi$	almost
MIN	yes

2.4.2. Asynchrony. Although the literature now includes a variety of calculi that everyone agrees are asynchronous, the definition of asynchrony remains informal. Roughly, a calculus is asynchronous unless a sender can observe that its message has been received without any cooperation from the receiver. Peter Selinger has made the boldest attempt to formalise this definition [**34**]. His approach is to identify redundancy in the labelled transition relation of a sort that must exist in order to hide information from senders.

Selinger's work is good, but the problem is far from solved. He considers three example systems that differ in the behaviour of the composition operator. For each he identifies sufficient and necessary conditions on the labelled transition system for the possibility of defining a *transparent process*, whose composition with any other process behaves just like the other process, up to weak bisimilarity. I have no doubt that this is the right criterion. As you might expect, the conditions depend significantly on the composition operator. Unfortunately, and despite tantalising commonality, the three sets of conditions have yet to be understood as members of a more general pattern.

The most complex of Selinger's examples is what one might call asynchronous CCS. I will leave you to guess what this calculus is, either by removing output guards from CCS, or by removing channel-passing from the asynchronous π-calculus.

[4]Equators are therefore a bit like fusions

The other two calculi both separate the channels into inputs and outputs. They differ in that the order of the messages on different channels is significant in one (the *queue* model) but not in the other (the *buffer* model). None of these is a channel-passing calculus, and as far as I am aware asynchrony conditions have never yet been worked out for a channel-passing calculus.

There are real practical benefits to be gained by finding asynchrony conditions for a calculus. Julian Rathke argues convincingly that redundancy in the labelled transition relation can be exploited to simplify the task of checking program equivalence [**30**]. Often, it makes the difference between an infinite and a finite calculation.

The current situation is a little unfortunate for MIN. Although it is undoubtedly asynchronous in the informal sense, MIN has a composition operator that differs from all three of Selinger's examples, and from the asynchronous π-calculus, and moreover MIN is a channel-passing calculus. I have struggled to achieve a partial understanding of MIN's asynchrony, which I have found valuable in many places, and indispensable in at least one, as you will see in chapter 3. Interestingly, it appears to be much closer to a mixture of Selinger's queue- and buffer-based asynchrony than it is to asynchronous CCS.

My partial understanding consists of no less than twelve conditions, which I know to be sufficient for asynchrony, but I do not know that some of them are necessary. Also, one depends on a complex auxiliary definition that classifies input transitions as necessary or superfluous.[5] In summary, the work is not yet ready to be published. However, I could not avoid including seven of the conditions (the tidier ones, fortunately) in chapter 3. Even as they stand, they push the boundary of this corner of research.

2.5. Functional Programming

For a while, there was a long list of features that were unique to functional programming languages. These included garbage collection, a formal semantics, strong types, algebraic types, referential transparency, and anonymous function closures, for example. These days most of these features have found their way into other kinds of language. Java has garbage collection and enough information hiding to support referential transparency, for example, and Python has anonymous functions. Approaching the divide from the other end, both ML and OCaml (which are functional languages) have mutable storage and a defined order of execution. Thus, the classification of certain languages as functional has become somewhat arbitrary.

One defining characteristic remains sharp, and it is a matter of design philosophy. On the one hand there are languages which rely for their efficiency of providing programmers with the means to direct the compilation minutely, and on the other there are languages, including functional languages, whose compilers rely on the mathematical tractability of the language. This divide is likely to remain sharp for ever, at least for high-performance languages, because a strategy between the two does not work, as Python or any other scripting language will witness [?]. MIN falls on the same side as functional languages: it relies on mathematical tractability. In this respect it is practically unique among languages designed for concurrency.

[5]This definition is very much along the lines of Fernández and Mackie's work on Interaction Nets, interestingly.

This is not the easiest way to design a programming language. You absolutely have to write a good compiler, or you will not get good performance. However, it is an attractive approach because many of the pessimisations necessary to write properly structured, maintainable programs are of a mathematically fairly simple form. I'm thinking of opaque types, templates, and liberal use of combinators, for example. This also applies to pessimisations such as null pointer checking that can be used to get important safety properties. Given a clever compiler, these desirable pessimisations do not cost anything.

2.5.1. Structure of a compiler. The strategy of relying on mathematical tractability implies a structure for the language's tool chain. After lexing and parsing, the first step in compiling a program is to expand the syntactic sugar, to obtain a term of a *core language*. For a functional language, this will be some sort of enriched lambda calculus. Type inference and checking is performed in the core language. The program is then heavily analysed and transformed, all within the core language. All interesting optimisations happen at this stage. The program then goes through a *lambda lifting* step, which I will describe below. Finally it is fed to a code generator that looks a lot like a C compiler. In the case of the Glasgow Haskell Compiler described in [**28**], the code generator actually is a C compiler.

Obviously there are variations on this pattern. GHC is primarily a batch compiler, but the overall structure can also be applied to a language that is based on a virtual machine. The higher-level stages of the compiler, down to and including the lambda lifting step, remain in the compiler. The file format which results then forms the executable format for the virtual machine. The virtual machine's JIT compiler then completes the compilation when the program is run.

MIN is a language based on a virtual machine, and it follows the above pattern. The high-level language has not yet been designed; that part of the project is not risky. The calculus called MIN that is the object of study of this thesis is intended to be the executable format of the virtual machine. This puts it *after* the lambda lifting step. In studying optimisations of MIN, I am implicitly exploiting the fact that the lambda lifting step is reversible. This variation has been used for functional languages too [**6**]. The language corresponding to MIN but *before* the lambda lifting step might look rather more like an ordinary π-calculus.

Thus one of my key innovations is an understanding of the lambda lifting step in the context of process calculi.

2.5.2. Lambda lifting. Restricting attention to functional languages for a moment, the purpose of the lambda lifting step is to cope with the substitution operation which appears in the definition of beta-reduction in the lambda calculus. If approached naively, substitution is the most expensive operation that the implementation will perform. Substitution also appears in the definition of the reduction relation of the π-calculus, and so lambda lifting is relevant to process calculi too.

The naive approach to implementing substitution is exemplified by the idea of compiling the lambda calculus into a term built from the S and K combinators, which is a famous failure. More generally, breaking every beta-reduction down into a sequence of smaller operations chosen from some finite menu is a mistake. The recommended approach to compiling a program [**27**] is instead to choose a different system of combinators, with similar properties to S and K, but tailored for that particular program. Each beta-reduction then maps onto a single beta reduction in the combinator reduction system, thus eliminating a lot of unnecessary control flow logic. This is what lambda lifting does.

2.5.2.1. *Example: quicksort.* I am aware that the lambda lifting algorithm may not be familiar to many readers, so I will illustrate it with an example. Here is a quicksort program, written in some imaginary functional language (a mixture of SML and OCaml, but allowing 'let' expressions to be recursive):

```
let quicksort gt =
  let qsappend acc [] = acc
    | qsappend acc (pivot::xs) =
      let partition gts lts [] = (gts, lts)
        | partition gts lts (x::xs) =
          if gt pivot x
          then partition (x::gts) lts xs
          else partition gts (x::lts) xs in
      let (gts, lts) = partition ([], []) xs in
      qsappend (pivot::(qsappend acc gts)) lts in
  qsappend [] in
quicksort;
```

As is typical of functional programs, the functions `partition` and `qsappend` have free variables in addition to their formal arguments. Specifically, `qsappend` mentions itself and the comparison function `gt` free, and `partition` mentions itself, `gt` and `pivot`. The first step is to remove these using beta-expansion. Note that it is possible to beta-expand from `partition` the whole of `gt pivot` at once.

```
let $quicksort gt =
  let $qsappend qsappend gt acc [] = acc
    | $qsappend qsappend gt acc (pivot::xs) =
      let $partition partition gt_pivot gts lts [] = (gts, lts)
        | $partition partition gt_pivot gts lts (x::xs) =
          if gt_pivot x
          then partition (x::gts) lts xs
          else partition gts (x::lts) xs in
      let partition = $partition partition (gt pivot) in
      let (gts, lts) = partition [] [] xs in
      qsappend (pivot::(qsappend acc gts)) lts in
  let qsappend = $qsappend qsappend gt in
  qsappend [] in
$quicksort;
```

The expanded functions are the ones whose names begin with a $. Each such definition is immediately followed by a 'let' expression that binds the old function name to an expression equivalent to its old definition.

The functions are now all combinators (that is, they have no free names), and can be promoted to the top level. The other 'let' bindings must stay where they are.

```
let $partition partition gt_pivot gts lts [] = (gts, lts)
  | $partition partition gt_pivot gts lts (x::xs) =
    if gt_pivot x
    then partition (x::gts) lts xs
    else partition gts (x::lts) xs in
```

```
let $qsappend qsappend gt acc [] = acc
  | $qsappend qsappend gt acc (pivot::xs) =
    let partition = $partition partition (gt pivot) in
    let (gts, lts) = partition [] [] xs in
    qsappend (pivot::(qsappend acc gts)) lts in
let $quicksort gt =
    let qsappend = $qsappend qsappend gt in
    qsappend [] in
$quicksort;
```

In fact, the functions are supercombinators (that's what the $ stands for).

DEFINITION 2.5.1. (Supercombinator)

A supercombinator is a combinator of the form $\lambda x_1. \dots \lambda x_n.E$ where all lambda abstractions in E are themselves supercombinators.

Supercombinators are especially easy to compile, for two reasons. First, all arguments must be supplied before the body can be instantiated; there's no point in doing any work until that moment. Second, the only information needed to instantiate the body is the values of the arguments. A supercombinator therefore compiles nicely into a single straight-line piece of machine code (in the simple case of no pattern-matching).

2.5.3. Correspondence of concepts.

- The π-calculus concept that corresponds to a lambda abstraction is an input prefix.
- The π-calculus transformation that corresponds to the well-known 'case-of-case' transformation of functional programs is the expansion rule.
- The MIN concept that corresponds to a supercombinator is a destructor.
- The MIN concept that corresponds to the body of the supercombinator is the collection of rewrite rules of the destructor.
- The MIN concept that corresponds to an argument of the supercombinator is a port of the destructor.
- The MIN concept that corresponds to the final (often pattern-matched) argument of a supercombinator is the principal port of the destructor.

Bibliography

[1] Andrea Asperti and Stefano Guerrini, *The Optimal Implementation of Functional Programming Languages*, Cambridge University Press (1998).

[2] Nick Benton and Andrew Kennedy, *Monads, Effects and Transformations*, ??? (1999).

[3] G. Berry, and G. Boudol, *The Chemical Abstract Machine*, Theoretical Computer Science, 96, pages 217–248 (1992).

[4] Gavin Bierman, Andrew Pitts and C.V. Russo, *Operation Properties of Lily, a Polymorphic Linear Lambda Calculus with Recursion*, Hoots (2000).

[5] Damien Doligez and Xavier Leroy, *A concurrent, generational garbage collector for a multi-threaded implementation of ML*, Proceedings 20th Symposium on Principles of Programming Languages (1993) pages 113–123.

[6] Marko van Eekelen, Sjaak Smesters and Rinus Plasmeijer, *Graph Rewriting Systems for Functional Programming Languages*, ??? (1996?).

[7] Maribel Fernández, *Type assignment and Termination of Interaction Nets*, Mathematical Structures in Computer Science (1995), volume 11.

[8] Maribel Fernández and Lionel Khalil, *Interaction Nets with McCarthy's 'amb': Properties and Applications*, Nordic Journal of Computing ??? (2003).

[9] Maribel Fernández and Ian Mackie, *Coinductive Techniques for Operational Equivalence of Interaction Nets*, Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science (1998) pages 321–332.

[10] Maribel Fernández and Ian Mackie, *Operational Equivalence for Interaction Nets*, Theoretical Computer Science 297 (2003) pages 157–181.

[11] Cédric Fournet, and Georges Gonthier, *The reflexive CHAM and the join-calculus*, POPL '96.

[12] Van Glabeek, *The Linear Time - Branching Time Spectrum* I, Report CS-R9029, CWI Amsterdam (1990).

[13] Kohei Honda, *The Asynchronous π-calculus, ??? (199?).*

[14] Kohei Honda and Nobuko Yoshida, *On Reduction-based Process Semantics*, TCS 152(2) pages 437–486 (1995).

[15] Kohei Honda, *Notes on Undirected Actions*, ??? (1995)

[16] Yves Lafont, *Interaction Nets*, Proceedings of the 17th ACM Symposium on Principles of Programming languages (POPL '90), pages 95–108. ACM Press, 1990.

[17] Yves Lafont, *Interaction Combinators*, Information and Computation, 137(1), pages 69–101, 1997.

[18] Xavier Leroy, Manifest types, modules, and separate compilation, Proceedings of the 21st Symposium on Principles of Programming Languages, pages 109–122 (1994).

[19] Xavier Leroy, *A syntactic theory of type generativity and sharing*, Journal of Functional Programming 6 (5): pages 1–32 (1996).

[20] Xavier Leroy, *The effectiveness of type-based unboxing*, Workshop in "Types in Compilation", Amsterdam (June 1997).

[21] Ian Mackie, *Interaction Nets for Linear Logic*, Theoretical Computer Science 247 (2000) pages 83–140.

[22] Ian Mackie and Jorge Sousa Pinto, *Compiling the λ-calculus into Interaction Combinators*, ??? (1998).

[23] Robin Milner, *Calculus of Communicating Systems*. Lecture Notes in Computer Science '92, Springer-Verlag (1980).

[24] Robin Milner, and Davide Sangiorgi, *Barbed Bisimilarity*, Proceedings of ICALP '92, Lecture Notes in Computer Science 623, pages 685–695, Springer-Verlag (1992).

[25] Uwe Nestmann and Benjamin Pierce, *Decoding Choice Encodings*, Basic Research in Computer Science RS-99-42 (1999).

[26] Joachim Parrow, and Peter Sjödin, *Multiway Synchronization Verified with Coupled Simulation*, Proceedings of Concur '92, volume 630 of LNCS, pages 518–533, Springer (1992).

[27] Simon Peyton Jones, *The implementation of Functional Programming Languages*, Prentice-Hall (1987).

[28] Simon Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain and Phil Wadler, *The Glasgow Haskell compiler: a technical overview*, Proceedings of the UK Joint Framework for Information Technology (JFIT) Technical Conference, Keele (1993).

[29] Jorges Sousa Pinto, *Sequential and Concurrent Abstract Machines for Interaction Nets.*

[30] Julian Rathke, *Resource-based Models for Asynchony*, ??? (???).

[31] Didier Rémy, and Jérôme Vouillon, *Objective ML: An effective object-oriented extension to ML*, Theory and practice of object systems ??? (1998).

[32] Davide Sangiorgi, *π-calculus, internal mobility, and agent-passing calculi*, (1995). Parts can be found in Proceedings of TAPSOFT ’95 and ICALP ’95.

[33] Davide Sangiorgi, and David Walker, *The π-calculus, A Theory of Mobile Processes*, Cambridge University Press (2001).

[34] Peter Selinger, *First-order Axioms for Asynchrony*, ??? (???).