CHAPTER 1

Introduction

There is a pressing need to write large, fast, reliable, portable programs cheaply and quickly. "Quickly" means that the task must require minimal effort, and "cheaply" means it must be performed by inexperienced programmers. To resolve these conflicting requirements, we rely on well designed programming languages and good optimising compilers.

For single-threaded programs, good languages and compilers are now readily available. We understand how to specify an interface between two parts of a program. We know how to find many common mistakes at compile time, before the program is ever run. We know how to find optimisations, and we know how to check that they do not change the meaning of the program. With these techniques, the gap between "quickly" and "cheaply" is readily filled.

For multi-threaded programs, we don't yet have any of this knowledge. It is frustratingly rare to find a program library whose use of concurrency primitives is sufficiently documented. The best way to avoid thread-related problems is to avoid using threads, or failing that to stare at your code long and hard. In the presence of threads, optimising compilers simply disable any optimisations that might not be valid. The result is that few programmers will risk using threads, and those that do work hard but still produce programs that are slow and unreliable.

This situation is alarming. Concurrency is very important in a wide variety of applications, as diverse as web browsers, database servers, operating systems, device drivers, and embedded controllers for telephones and aeroplanes. At the moment, such software is simply expensive.

1.0.1. Contribution. This thesis addresses the problem by improving our understanding of the meanings of multi-threaded programs, and the implications of decisions in the design of multi-threaded languages.

My principal contribution is a rather unusual calculus called MIN, which is designed to form the basis of a useful programming language, suitable for writing device drivers and web browsers, while at the same time remaining mathematically tractable. I develop MIN's theory to the point at which validating an optimisation is a fairly automatic task, and I give an encoding into MIN of a realistic high-level language with threads, locks, IO, updatable storage, and objects with proper information hiding, subtyping and inheritance. This overlap of theory and practice is very encouraging.

Of course, it is not possible to solve all industry's problems in the course of a single PhD, and there are crucial aspects that I have not addressed at all. I have only a sketchy idea how to compile MIN efficiently into machine code. Consequently, I don't yet know how to tell if a MIN program is efficient, let alone what strategy to use to optimise it. I am certain, however, that a good core calculus, backed up by a solid bit of theory, is a prerequisite for studying these problems.

1.1. Theory and Practice

Most of the mathematical techniques I need already exist, and have been applied to a succession of process calculi over a period of more than twenty years. It is surprising how little impact these ideas have made on the way languages are designed and compilers are written. In fact, I will go so far as to say it is depressing.

The wall that separates the theoretical and practical sides of computer science seems to be discouragingly permeable to complexity and detail, while holding back all the big ideas. Nobody seems to want to go near it. In writing this thesis I have tried to break through the wall, to engage with the detail, and to understand both sides at once, so as to be able to build a better wall.

1.1.1. An abstraction layer. The best walls are successful models. For example, the transistor is an excellent model that separates those who design integrated circuits from solid-state physicists; the band-gap model of a semiconductor separates solid-state physicists from chemists; the atom separates chemists from nuclear physicists; and so on. Each model is used as a common language between neighbouring disciplines in which to phrase requirements and describe achievements, while hiding everything else. In programming terms, models are the abstraction layers used to organise human knowledge.

What is the model that separates theory and practice in computer science? It is certainly not the lambda calculus, which for a while has been the wall between theoretical computer scientists and purer mathematicians. Nor is it assembler, which forms the wall between practical computer scientists and those who design integrated circuits. Currently, it is something approximating the programming language C. Theoreticians (I include those who study functional languages) consider a problem completely solved if it can be reduced to a program in a C-like language, and compiler-writers see their task as compiling C programs.

I feel that the use of C-like languages for this purpose is a mistake. Such languages do not form a good model of computation for several reasons. They are complicated. They are single-threaded. They are often incompletely specified, so that the abstraction layer has holes, and does not form a good separation of the disciplines. The problem of compiling C is already almost perfectly solved. However, the most persuasive failings are the ones that are pragmatic and immediate, such as the complete failure of the last few decades of theory to make any impact at all on the design of processors. There is just no way that a C-like language can express ideas about functional languages or process calculi.

I am encouraged by a recent trend towards safe, garbage-collected languages. In many ways, Java is much nearer to a functional language than to C, and recent efforts to adapt processors specifically for running Java are a good sign. However, Java is not a functional language, and I don't think the trend will go that way. A pure functional language would also be a bad model of computation. Pure functional languages do not model IO well. They can't express imperative update. They are still single-threaded.

1.1.2. Purpose. I would like to see computer science adopt a language like MIN as an abstraction layer. It is simple compared to the fields it separates. It is formally specified, so it will form a good seal against irrelevant details. Its model of IO is close to the physical reality, so it can express good abstractions of things like disks and network cards. It can express imperative update. Most immediately, it is multi-threaded, so it can express ideas about concurrency and determinism that are not currently getting through.

I would like to see the practical side of computer science treat MIN or a similar language as the problem they need to solve. Compiling MIN into C and then building a processor for running C is certainly not the best way of running MIN, even if both steps were performed perfectly, because there are opportunities for optimisations in MIN that are impossible in C. For example, a MIN program exposes a great deal of parallelism, as well as useful information about locality and representation of data, and opportunities for removing non-determinism, all of which would be lost by translating into C.

I would like to see the theoretical side of computer science treat MIN or a similar language as an abstraction of what real computers can do. To compile a language all the way down to C is to repeat a great deal of work that is essentially the same for every language; better to stop at MIN, and leave the remaining work to the specialists. MIN is ideal for this purpose. It is simple and formal and portable and tractable, so the effort invested will be protected from the relentless march of technology, and it is able to express plenty of high-level information about parallelism, locality, representation of data, and opportunities for removing non-determinism, that can be used for platform-specific optimisations.

1.1.3. Organisation of the text. I have therefore divided this thesis, apart from this introductory chapter and the concluding chapter 7, into two parts.

Chapters 2 to 4 are designed to be comprehensible and appealing to theoreticians. It is this theoretical part that is the best developed. I explain where all of the mathematical techniques have come from, how I have applied them to MIN, and what results I have achieved. Principally, I develop a technique for proving that two programs are equivalent, and I establish beyond doubt that the technique is both sound (it works) and complete (no other technique is necessary). Chapter 4 incorporates the important effect of types.

Chapters 5 and 6 are designed to sell the idea to more practically minded people. Ideally, I would have produced evidence for my belief that an efficient implementation of MIN is possible, but I have not had time to perform any experiments. Instead, I offer an argument that such evidence would be valuable, by showing in detail how it would be immediately applicable to a realistic high-level language. My example language is not real, but it is comprehensive: I am eager to show that I have not needed to make any gross sacrifices to the god of mathematics.

1.2. MIN

An example MIN program is shown in figure 1.2.1. One unusual feature of MIN that will be immediately obvious is that it uses a graphical syntax. The syntax is formal and precise. I have also invented a textual syntax, which provides a convenient way to enumerate all programs, and which I therefore use in chapter 3 to prove things about MIN, but the graphical syntax is the one I consider primary, and which I use for all programming. I will discuss this feature properly in section 1.4.

The program is structured in three main sections. The *main graph* plays the role of the 'main()' routine of a C or Java program, but it also represents the entire initial state of the program: it is the data that the program manipulates. The *rewrite rules* are the code. The third section consists of *node declarations* and *subtype declarations*, along with an *interface* in the form of the annotations on the main graph. These

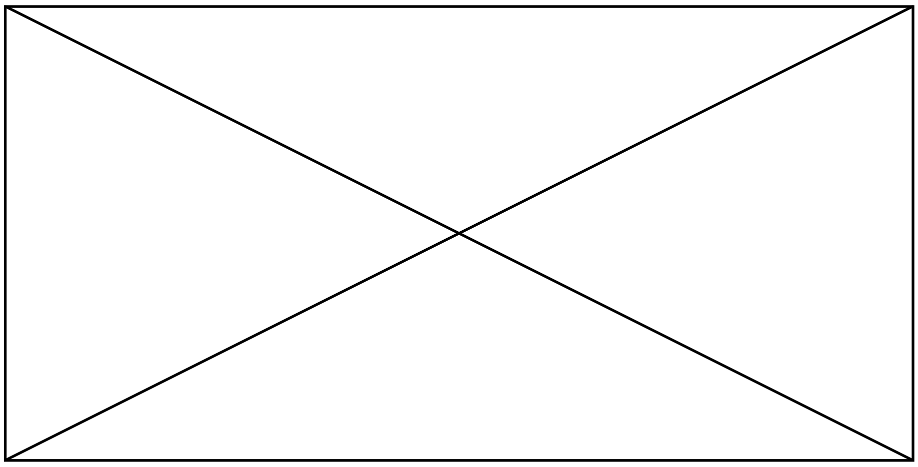

FIGURE 1.2.1. *[1A]* A MIN program that implements an AND gate.

define type-checking restrictions on the way in which the program may be linked. All MIN programs have these three sections. If ever I draw a graph on its own, it is the main graph of a program, and its rewrite rules and declarations will be in a nearby figure.

I give a precise and complete language definition in section 1.3. Feel free to read it first if you want. The rest of this section provides plenty of information about MIN at an informal level. It is my attempt to hook you without first asking you to invest effort in learning MIN.

1.2.1. What sort of language? The program in figure 1.2.1 is an implementation of a logical AND operation. The way in which it tackles this simple and familiar task illustrates some of the unusual adaptations MIN has for its application.

As you would expect in any language, most of the very short program is spent defining the truth table of an AND gate. However, MIN is a very simple language and it doesn't have 'operator' or 'expression' as primitive concepts. The program has to define explicitly that if booleans are attached to the ports x and y then shortly afterwards a boolean will emerge from the port z. It also includes a definition of 'boolean', which will be checked for consistency with other programs during linking. In this way MIN is verbose and pedantic, rather like assembler. MIN is a low-level language, and user-friendliness is one of its lowest design priorities.

Unlike assembler, however, MIN is exceptionally precise. Given implementations of OR and NOT gates similar to the AND gate, it is possible to prove all of the equivalences in figure 1.2.2 (and essentially all the axioms of Boolean algebra) *without any further assumptions*. To do the same trick in assembler would require assumptions about, at the very least, the calling convention, the memory allocation convention, and — given that we're in a multi-threaded world — the convention obeyed by interrupt (or signal) handlers. This is particularly bad because none of those conventions holds universally. To put the point in practical terms, the optimisations in figure 1.2.2 (for all program equivalences are potential optimisations) are possible in MIN but not in assembler.

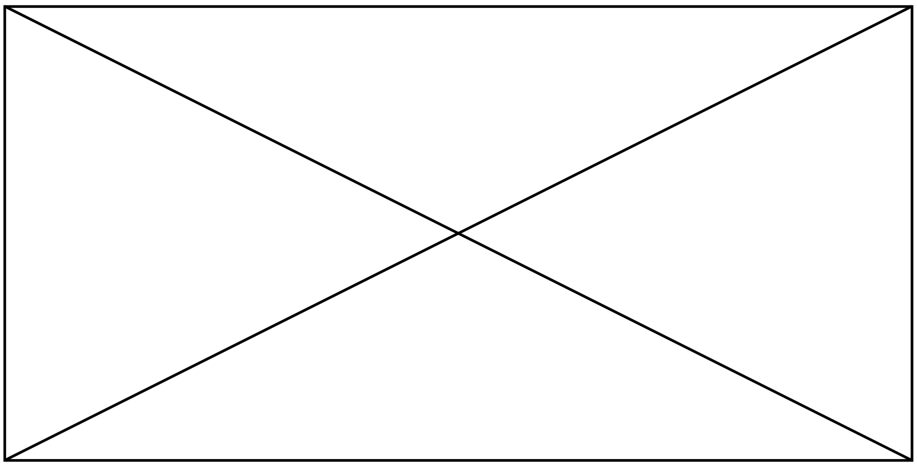

FIGURE 1.2.2. *[1B]* Some program equivalences involving the program in figure 1.2.1 and some similar programs. These can be used to optimise programs.

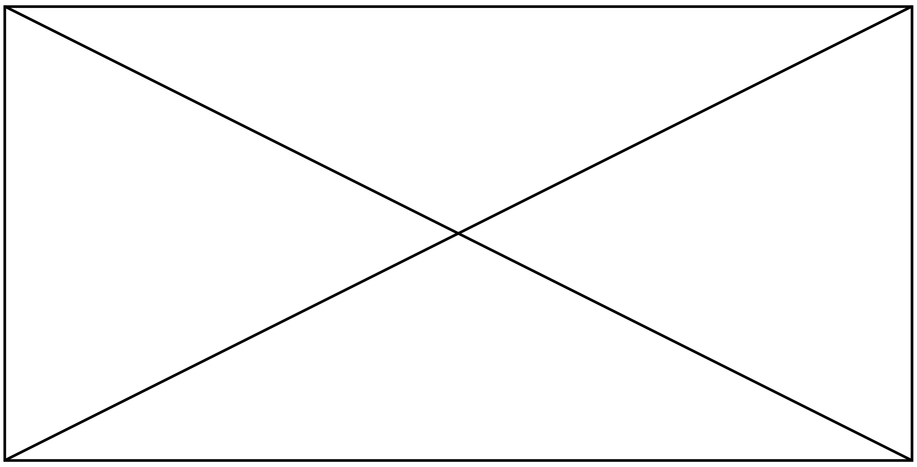

FIGURE 1.2.3. *[1C]* An implementation of the 'parallel AND' operation, which returns 'false' if either of its inputs is 'false', even if the other input does not arrive, and returns 'true' only if both inputs are 'true'.

These program equivalences can also easily be proved in functional languages, but MIN is not a functional language. Functional languages make many more assumptions that MIN manages to do without. For example, in MIN it is easy to implement 'parallel AND' (figure 1.2.3), and to prove the same sort of equivalences about it as about the ordinary AND gate, but this is already beyond what a functional language can express. More impressively, MIN is perfectly capable of expressing interrupt handlers, schedulers, locks, and all the other dirty violations of the usual conventions (the very violations that trip up attempts to reason about assembler). These components are thoroughly non-functional.

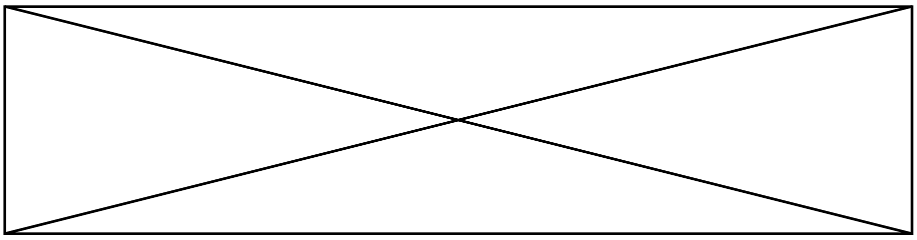

FIGURE 1.2.4. *[1D]* A program that can be linked with that in figure 1.2.1.

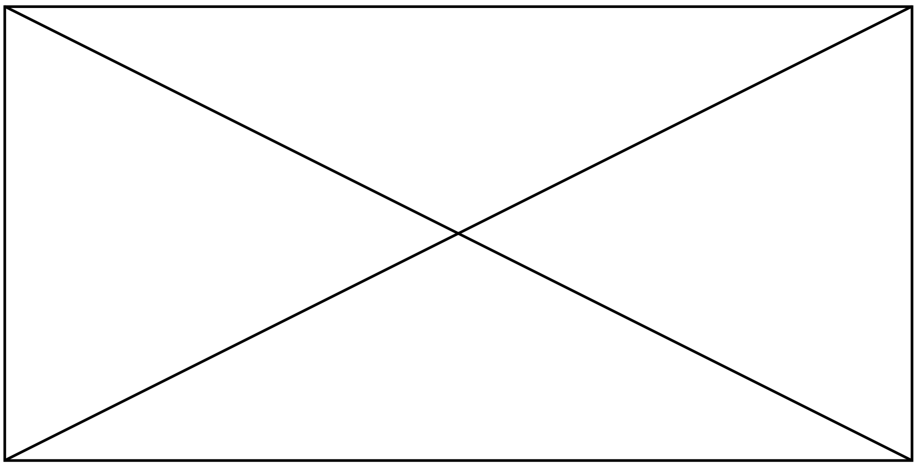

FIGURE 1.2.5. *[1E]* The program that results from linking the programs in figures 1.2.1 and 1.2.4.

The trick is to write down all the assumptions in every program; let nothing be tacitly assumed. Mechanisms for documenting programs formally are becoming common in programming languages, but only in high-level languages, where they are used to organise and debug programs that are so large that no programmer could be expected to remember how they work. To find these mechanisms in a low-level language like MIN is quite unusual, but it is essential for my chosen application: a compiler cannot do the job of a programmer unless it has access to the same information.

In summary, MIN is a rather strange sort of language. It it low-level like assembler, and has the expressive power of a multi-threaded language, but it is formal and tractable like a functional language, and has interfaces and information hiding like an object-oriented high-level language.

1.2.2. Execution model. So how does the program in figure 1.2.1 work? It is designed to operate on data from other programs, and on its own it doesn't do anything. To see it in action, we must link it with a program that supplies some inputs, such as that shown in figure 1.2.4. In this program, I have deliberately called the type 'truth' instead of 'boolean' to show that I can; the linker works out that the two types are compatible. The result of linking the two programs is shown in figure 1.2.5.

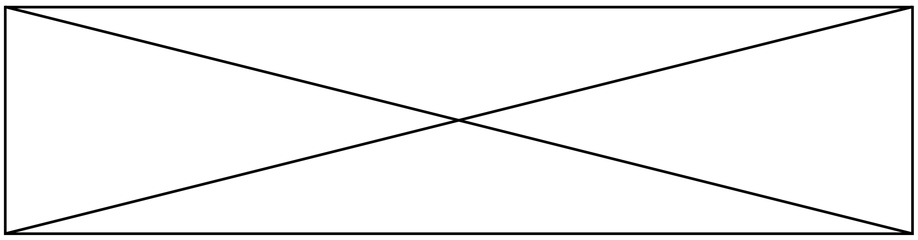

FIGURE 1.2.6. *[1F]* The unique sequence of two rewrites that the program in figure 1.2.5 can perform. Each of the intermediate states is a program. Only the main graphs are shown; the declarations and rewrites do not change.

The program can now perform some computation. Each step consists of locating in the main graph an occurrence of the left-hand side of a rewrite rule, and replacing it with the right-hand side. Only the main graph changes; all other parts of the program are static. Figure 1.2.6, which shows only the main graphs, starts from the program of figure 1.2.5 and performs a sequence of two rewrites. In this example, no other execution path is possible, but in general there is quite a lot of freedom; imagine substituting figure 1.2.3 for figure 1.2.1, for example. The final program is again blocked; it is of a form that can supply data to another program after further linking.

Thus, the two main concepts in MIN's execution model are linking and rewriting. Typically linking occurs first, but MIN is designed to be a dynamically compiled language, so sometimes running programs need to be linked too. The formal definition of when two programs are equivalent depends only on their linking and rewriting behaviour. Any detail of a program that cannot be discovered by linking and rewriting is hidden, and may be changed in order to optimise the program. Any detail that can be discovered must be preserved, as it forms part of the meaning of the program.

I would not like you to think that MIN's rewriting semantics must be implemented to the letter. It sometimes requires several MIN steps to achieve something that is primitive on real processors, such as dereferencing a pointer. A good MIN compiler will need to recognise idioms that permit special implementations, such as places where it could use a pointer, and it will need to be able to manipulate the program algebraically in order that the idiomatic tricks apply. It is also important that the compiler can throw away execution paths, in order to make the program more deterministic, because deterministic programs typically permit more efficient implementations than non-deterministic ones. The compiler's freedoms form an important part of the language specification.

1.2.3. Linking. The result of linking two MIN programs is another MIN program. The four kinds of identifier in a MIN program (names, constructor symbols, destructor symbols and types) all behave differently for the purposes of linking. Their behaviours illuminate their relationships with concepts in other programming languages.

Names: are linear: if two programs have a free name in common, then the name becomes bound when they are linked. This binding is analogous to a function binding, in the sense that bound names can be alpha-converted. However, the operation is symmetrical; it is not possible to say which program is underneath the binder. The purpose of names is to form connections between the two programs.

Destructor symbols: are private: if two programs have a destructor in common, then it is not the same destructor. One of them must be alpha-converted to something fresh in order to link the programs. The resulting program will contain both destructors, under different names.

Constructor symbols: are public constants: if two programs have a constructor in common, then it is the same constructor. When they are linked, the resulting program will just have the one constructor. Constructors form the common language that programs use to send messages to each other.

Types: are private, like destructors, in all respects except that types from any two programs can be compared using a subtype relation. In particular, two types with structurally identical definitions are each a subtype of the other. This allows different programs to be type-compatible even though neither can mention the other's type identifiers explicitly.

Types are a bit of a special case: their behaviour in MIN simply imitates their behaviour in many other programming languages. However, the disciplined separation of the other three kinds of identifier is a deliberate contribution to MIN's precision. MIN can encode languages whose constructs mix up connections, private values and public values, and moreover can encode them in a form in which optimisations remain possible. However, the converse is not always true: languages which do not use identifiers in a disciplined way may not admit a good encoding of MIN or of other languages that MIN can encode.

Although the four kinds of identifier all behave differently, they have one important property in common: they do not suffer from namespace pollution. There is no way in which an unfortunate coincidence of identifiers can cause two programs to interfere with each other. In other languages, this sort of interference, however improbable, can make the difference between a valid and invalid optimisation.

1.2.3.1. *Rewrite rules belong to destructors.* There is a syntactic restriction on the form of the left-hand side of a rewrite rule in MIN: it must consist of a single constructor and a single destructor. Given a particular constructor symbol and a particular destructor symbol that occur in a large program formed by linking many smaller programs, it is interesting to ask which of the smaller programs could possibly define a rewrite rule for the pair. The constructor symbol is a constant, and is therefore in scope in all of the smaller programs. In contrast, only one of the programs contributes the destructor symbol, and so only that program can possibly define a rewrite rule.

It is therefore sensible to think of rewrite rules as defining the behaviour of the destructor, and to think of constructors as not having any behaviour of their own. This is illustrated in figure 1.2.7, which shows two symmetrically related programs, the result of linking them, and the result of running the linked program. The example shows how the asymmetry between the linking behaviour of constructors and destructors leads to an asymmetry in the main graph.

1.2.4. Type system. Untyped MIN is a perfectly self-consistent language, but it is not nearly as expressive as typed MIN. As in other languages, there are many program transformations (that is, potential optimisations) that are valid in typed MIN but not in untyped MIN. This is quite counter-intuitive, because the types are erased at run-time, and have no effect on the rewriting behaviour of a program.

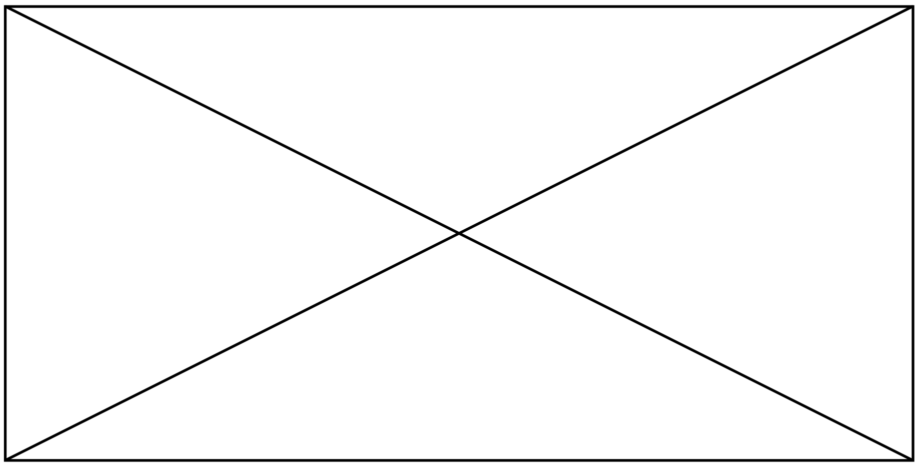

FIGURE 1.2.7. *[1H]* Linking does not respect constructor-destructor symmetry. (a) and (b) show two programs that differ only by exchanging all constructors to destructors. Declarations are omitted. (c) shows the main graph of the program that results from linking (a) and (b), and its reaction. The final graph is not symmetrical, as it only contains a constructor.

The effect of types on the meaning of a program is mediated only by linking: types can prevent certain programs from being linked together.

I think there are two sweet spots on the spectrum from no types to full proof-carrying code. One of them is full proof-carrying code, capable of expressing anything that a programmer can think, and in particular capable of encoding the type scheme of any other language. That is beyond the scope of this thesis. The other is a 'simple' type system, as in the simply-typed lambda calculus. This thesis includes a whole chapter (chapter 4) about the effect of adding a simple type system, and it is used in all the example MIN programs.

In many languages the type of a variable is interpreted as the set of values that the variable may have. Some types in MIN can be interpreted in that way, but 'value' is not a very useful concept in MIN. We need another picture. Constructors can be seen as messages sent from one part of a program to another, and wires as the channels along which these messages travel. Types in MIN are annotations on the wires of a program which impose a restriction on the patterns of messages that can travel along them. In other words, a type is a state of a communication protocol.

The protocols definable even in the simple type system can be quite complicated. In any particular state of a protocol, one party is the sender and the other is the receiver, and there is a finite menu of constructors that the sender can send. If and when a message is sent, the protocol may enter a new state, in which the menu is quite different, and the roles of sender and receiver may be interchanged. Moreover, protocols can branch: sending a message may result in two or more new wires, each with a different type. Only in the special case in which the sender and receiver are never interchanged in any future state of the protocol (that is, in which all messages flow in the same direction) can we view the sender as a value to which the receiver has access.

1.2.5. Linearity. You may have noticed that the execution sequence in figure 1.2.6 consumes the program, so that it cannot be used again. There are two things going on here, one of which is standard, but the other of which is quite unlike almost every other programming language.

First, the definition of the 'and' destructor is private to the program, by which I mean that another program cannot create 'and' nodes just by being linked with this one. This sort of local definition is found in many programming languages. For example, in OCaml we can perversely write

```
let and u v = match u with false -> u | true -> v
in and
```

to define a logical AND operation which is only locally called 'and'.[1] These lines of code do not add anything to the environment. This is standard stuff.

However, in OCaml, the anonymity of the function does not prevent us using it more than once. The expression might only occur once in a program, but if it occurs in a 'let' construction, or as an argument to a function, it will acquire a name that enables it to be shared.[2] The 'and' destructor in the MIN program is also accessed via names: x, y and z, but in contrast to the OCaml program each of these names can only be used once. I don't just mean that using a name again results in an error, I mean that a program that uses a name twice is syntactically incorrect, and won't even compile. In summary, names in MIN are *linear*.

My decision to use a linear language is sufficiently unusual as to warrant the whole of section 1.5.

1.2.6. Arbitration. Non-determinism is an essential feature of a multi-threaded language, but it is not enough: we need arbitration. A program must be able to make a decision according to which of two signals it receives, choosing non-deterministically only if it receives both. For example, a good user interface for a long-running calculation will provide a 'cancel' button. If the button is pressed before the calculation completes, the calculation is abandoned. If the calculation finishes first, the button vanishes and the result is displayed. If the button is pressed at the same moment that the calculation finishes (by which I mean that the two events are separated by a shorter time than it takes the computer to respond, which is quite probable if the computer is busy), either could happen.

Just as it is not possible to write a non-deterministic program without a non-deterministic primitive, it is not possible to write an arbiter without an arbitration primitive. MIN therefore has an arbitration primitive. Specifically, it is possible to define rewrite rules with left-hand sides that can overlap. Figure 1.2.3 shows an example.

However, it is important to ensure that whenever a set of redexes overlaps in pairs, they also all have a common overlap. More precisely, if A overlaps with B and B overlaps with C then A, B and C must have a common overlap. Any arbitration primitive without this property will be difficult to implement efficiently, as it will occasionally be necessary to reach a widely distributed consensus. As long as there is a node involved in all of a set of competing redexes, that node can perform the arbitration efficiently and locally.

[1]Actually, 'and' is a reserved keyword, but we could call it 'frog' instead, and it would make no difference.

[2]In a functional language, 'shared' means 'copied or discarded'.

MIN achieves this property using a syntactic restriction. Say a port of a node is a *principal port* if it is joined to another node in the left-hand side of some rewrite rule. The restriction is that constructors may only have one principal port. Combined with the rule that a redex must consist of one constructor joined to one destructor, this ensures that whenever two or more redexes overlap, the destructor belongs to all of them, and can therefore be the arbiter.

1.2.7. Matching. In any rewriting system, if the left-hand side of every rewrite rule is singly connected, then the net effect of any number of steps will be to replace one or more singly connected graphs with new graphs.[3] It is therefore impossible to write a program that behaves differently depending on whether or not the original graph is singly connected. Many languages therefore have a primitive that can detect double connections; let us call this a *match* primitive, after the famous example from the π-calculus. The match primitive in languages like Java and ML is equality testing of references.

MIN does not have a match primitive, and it therefore can't encode the above languages completely. This is a failing. I omit pointer equality testing from the Java-like language in chapter 5 precisely for this reason. Given the number of real languages that do have match primitives, it would be valuable to add one to MIN, but I fear this would significantly complicate the theory. In the longer term, it may be more productive to omit match primitives from new programming languages; although useful for clever optimisations, a match primitive does not appear to be necessary for normal programming. More research is needed in this area.

1.2.8. Misconceptions. *[Where should this go? Here? After section 1.2.1? Nowhere?]*

I would like to dismiss two misconceptions that I find are common among programmers and computer scientists. If you find you suffer from either ailment, you will probably not understand why MIN is the way it is.

1.2.8.1. *Adding features does no harm.* From a programmer's point of view, a feature of a programming language that exists does not appear to do any harm to a program that does not use it. However, this viewpoint is flawed.

For example, referential transparency is an important part of a pure functional language, which both helps programmers to understand each others' code and also helps compiler optimisations. Adding an impure feature to a pure functional language would add an additional burden to all programmers and compilers even when they are engaged with a program that is pure functional, for two reasons. First, it may take effort to be sure that the program is pure functional, and the lingering doubt is draining. Second, the program may one day be linked with one that is not pure functional.

[3] I have made an oversimplification here. The true version of this statement is more complicated. Given an undirected graph we can unroll it into a possibly infinite singly-connected graph as follows.

Choose one of the nodes of the input graph and call it the root (up to graph isomorphism the answer will be the same no matter which node is chosen as the root). The unrolled graph has a node for every path through the input graph from the root node, and an edge for every step that can be appended to such a path.

This construction has the property that any singly-connected subgraph of the input graph appears unchanged in the unrolling, but copied possibly infinitely many times. Also, unrolling commutes with rewriting (provided we allow ourselves to rewrite all copies at once). There is a sense in which the unrolled graph can simulate the input graph. Therefore the input graph cannot tell that it is not its unrolling.

In general, adding a feature often removes useful assumptions. This in turn makes programming more difficult, and also invalidates optimisations. The result is that programs become less reliable and less efficient.

1.2.8.2. *All that matters is the compiled code.* Programmers often regard their job as being the production of executable code. The language which they use, and the style in which they write their programs, is nobody's business but their own. All these decisions are stripped out or hidden by the compiler, and all the user sees is a piece of machine code. This viewpoint is flawed.

This misconception often appears in a subtly different form: translating a program from one language to another preserves everything if it preserves its behaviour when run. This is also wrong.

By taking this viewpoint to its extreme, we would conclude all of the following:

- There is no point in documenting code.
- Physical markup is equivalent to logical markup.
- There's no reason to use a safe language.
- Static types in programs are meaningless.

Clearly all of these conclusions are ridiculous. The flaw is the assumption that all one would ever want to do to a piece of source code is to compile it. There are many other things one might want to do, including fixing bugs (not to mention determining whether somthing is a bug or not), incorporating it into a larger program (taking account of any provisions made for future versions of the code), splitting it into modules or otherwise refactoring it (without breaking any security properties), changing or adding bits, porting it to another platform, handing it over to another programmer, and so on. For these purposes a great deal of extra information about the program is required in addition to its behaviour when run, and this extra information all forms part of the meaning of the program.

1.3. Language Definition

This section gives a precise and complete definition of MIN. The definition is short: it finishes on page 27. It consists of four ingredients:

- the set of syntactically correct programs,
- the reaction relation, which represents the rewriting behaviour,
- the operation of linking two programs, including the type-checking rules, and
- the freedom of implementation granted to the compiler.

Obviously all four are necessary, despite the preponderance of real languages whose definitions stop after the first two or three.

1.3.1. Grammar.

There is currently no widely agreed conventional notation for presenting a graphical grammar, analogous to Bachus-Naur form for a textual grammar. I hope this will be remedied soon. In the meantime, I have resorted to precise English, supplemented with plenty of examples. You can always refer to the textual notation defined in chapter 3 if my informal style leaves you feeling uneasy, but I hope you will not find it necessary. I will present the grammar from the bottom up, starting with the basic symbols that can be drawn on the page, and then defining progressively more complex structures, culminating in the definition of a whole program.

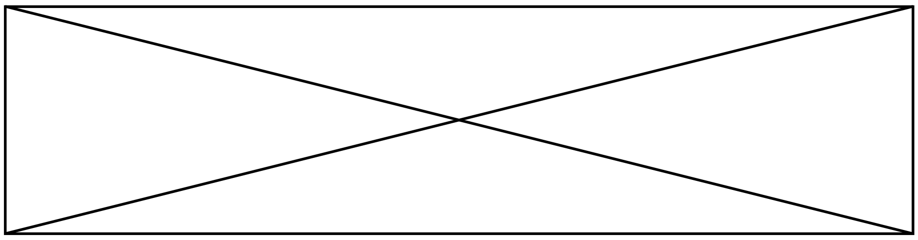

FIGURE 1.3.1. *[1M]* Some halt symbols.

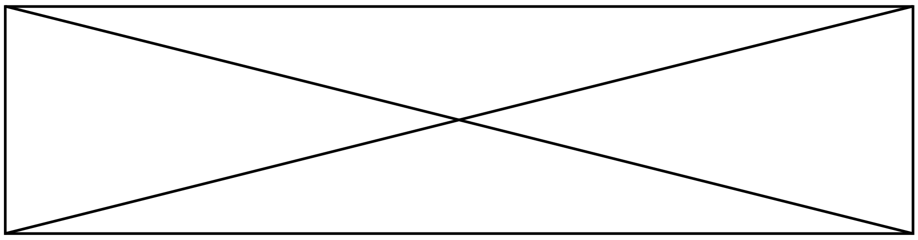

FIGURE 1.3.2. *[1K]* Some constructor symbols.

1.3.1.1. *Halt symbol.* A *halt symbol* represents a blocked (inert) program. It is drawn as a small circle with one or more 'legs' called *ports*. Halt symbols are equal iff they have the same number of ports. Moving a halt symbol around the page or permuting its ports makes no difference. Figure *1.3.1* shows some examples.

1.3.1.2. *Constructor symbol.* A *constructor symbol* represents a data value. It is drawn as a shape with a single closed outline, possibly decorated in some way, with one or more ports. One of the ports, called the *principal port*, must emanate from a corner of the shape. The others, called *auxiliary ports*, must emanate from flat sides. Constructor symbols are equal iff they have the same shape, decoration and tuple of ports. Rotating a constructor symbol or moving it around the page makes no difference, but reflecting it or otherwise permuting its ports yields a different symbol.

Typically, my constructor symbols are triangles containing a word or a letter. I usually draw them with the principal port pointing upwards, and I arrange the auxiliary ports along the opposite side. Neither convention is required.

Figure *1.3.2* shows some examples.

1.3.1.3. *Destructor symbol.* A *destructor symbol* identifies a piece of code. It is drawn as a shape with a double outline, possibly decorated in some way, with one or more ports. Those ports which emanate from corners are called principal ports, and the rest are auxiliary ports. There must be at least one principal port. Destructor symbols are equal iff they have the same shape, decoration and tuple of ports. Rotating a destructor symbol or moving it around the page makes no difference, but reflecting it or otherwise permuting its ports yields a different symbol.

Typically, my destructor symbols are crown shapes, with one or more sharp points for the principal ports, opposite a flat base along which I arrange the auxiliary

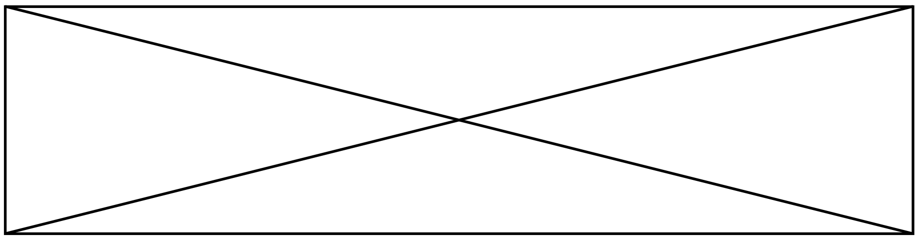

FIGURE 1.3.3. *[1L]* Some destructor symbols.

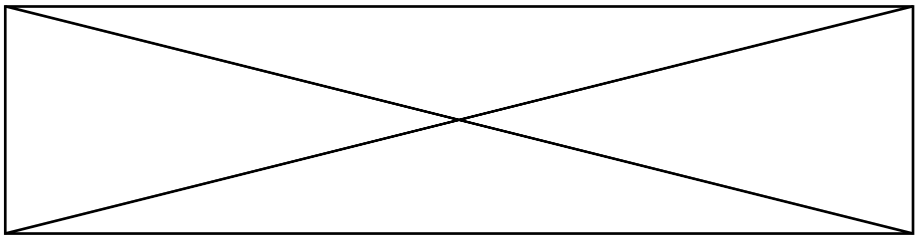

FIGURE 1.3.4. *[1N]* Some examples of things that are not node symbols. (a) A halt symbol must have at least one port. (b) and (c) A constructor symbols must have exactly one principal port. (d) A destructor symbol must have at least one principal port. (e) The orientation of this destructor symbol is ambiguous. (f) A node symbol must have a closed outline.

ports. When there is only one principal port, the crown is just a triangle. I usually draw the principal ports pointing downwards. These conventions are not required.

Figure *1.3.3* shows some examples.

1.3.1.4. *Symbol.* A *symbol* (somtimes called a *node symbol*) is a constructor symbol, a destructor symbol or a halt symbol. Figure 1.3.4 shows some things that are not symbols.

1.3.1.5. *Node.* A *node* is an instance of a symbol. Nodes are the unit of memory allocation. Nodes are called *constructor nodes*, *destructor nodes* or *halt nodes* according to their symbol. Nodes are equal iff they have the same symbol and the same position and orientation on the page; two different nodes can have the same symbol. The ports of a node are the instances of the ports of its symbol.

1.3.1.6. *Wire.* A wire is a bi-directional communication channel. It is drawn as a line, with no sharp corners, and no junctions. Wires have two ports, one at each end. The path that the line describes across the page is of no significance, except in that it should not go through nodes or ports, and that wherever it crosses other wires it should do so at a significant angle. Figure *1.3.5* shows some examples, and figure *1.3.6* shows some things that are not legal wires.

1.3.1.7. *Junctions.* Ports can be *joined*. All ports look like the end of a line; to draw two joined ports, simply join them end-to-end without a corner. No more that two ports may be involved in a junction (that is, each port may be joined to at

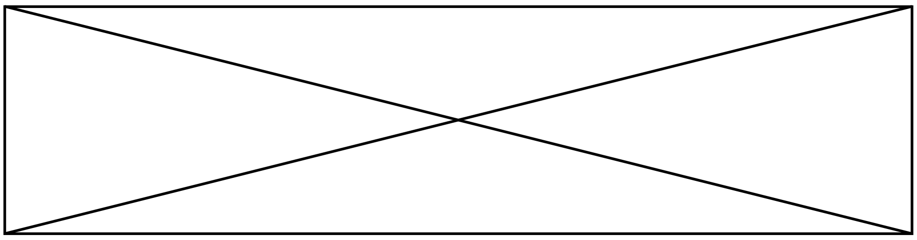

FIGURE 1.3.5. *[1I]* Some examples of wires

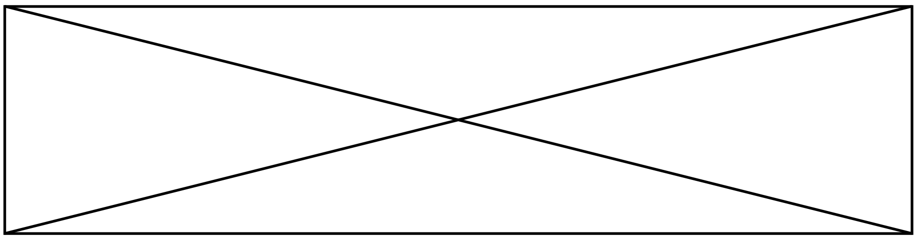

FIGURE 1.3.6. *[1J]* Some examples of things that are not wires.

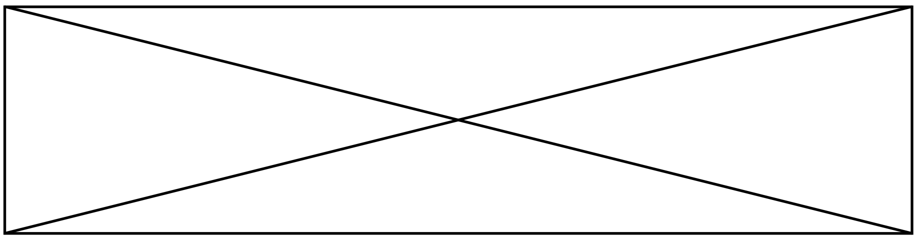

FIGURE 1.3.7. *[1S]* Some examples of junctions. (a) A port of a node joined to one end of a wire. (b) Two wires joined end-to-end, or it is three? The notation is deliberately ambiguous. (c) A wire joining two ports of the same node. (d) A port of one node joined to a port of another. (e) Two ends of a wire joined to each other to make a loop.

most one other port). Examples are shown in figure 1.3.7, and figure *1.3.8* shows some illegal junctions.

Note that it is possible to join together the two ends of a wire. The result is called a *loop*.

1.3.1.8. *Free port.* A *free port* is a port that is not joined to any other port. A free port must be annotated with a lower-case letter, called its *name*.

1.3.1.9. *Untyped graph.* An untyped graph represents the run-time state of part of a program. It consists of a finite set of nodes and wires. The ports of the nodes and wires may be joined to each other, but not to anything outside the graph. The names of the free ports must all be distinct, and are called the *free names* of the

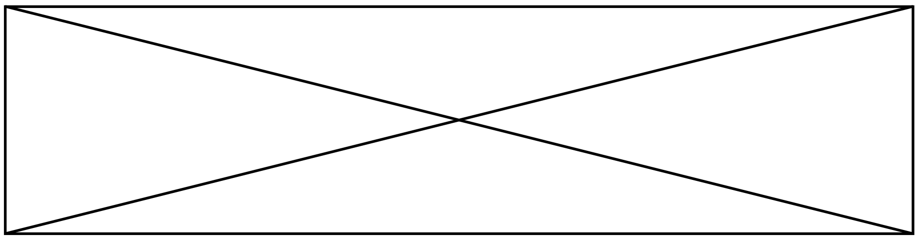

FIGURE 1.3.8. *[1T]* Some things that are not junctions. (a) Junctions must be smooth. (b) Only two ports may meet at a junctions. (c) Only ports can be joined, and there is no port in the middle of a wire.

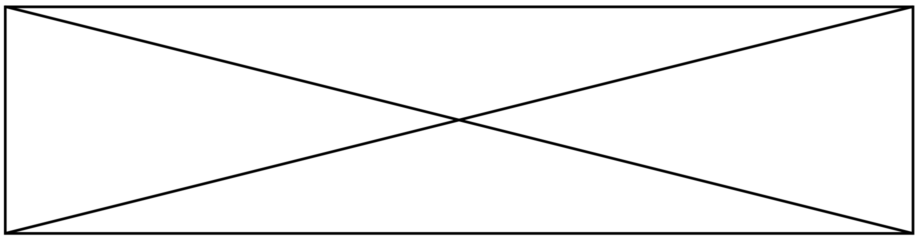

FIGURE 1.3.9. *[1O]* Some examples of legal graphs. The whole diagram is also a legal graph.

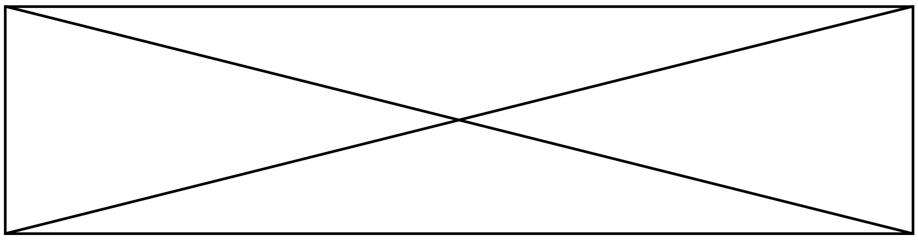

FIGURE 1.3.10. *[1P]* Some illegal graphs.

graph. Figure *1.3.9* shows some legal graphs, and figure *1.3.10* shows some illegal graphs.

The notation for graphs is deliberately ambiguous. The ambuguity is completely described by the following statement: whenever two ports appear to be joined, it is not possible to tell whether they are actually joined, or whether there is a chain of wires in between. In particular, two wires joined end-to-end are indistinguishable from a single long wire. Graphs which differ only by inserting or removing wires in this way are defined to be equal.

There are other ways in which graphs can be equal too. The full definition of graph equality is as follows: Two graphs are equal iff, after exploiting the ambiguity in reading the wires, it is possible to construct a bijection from the ports of one graph to the ports of the other, satisfying the following properties:

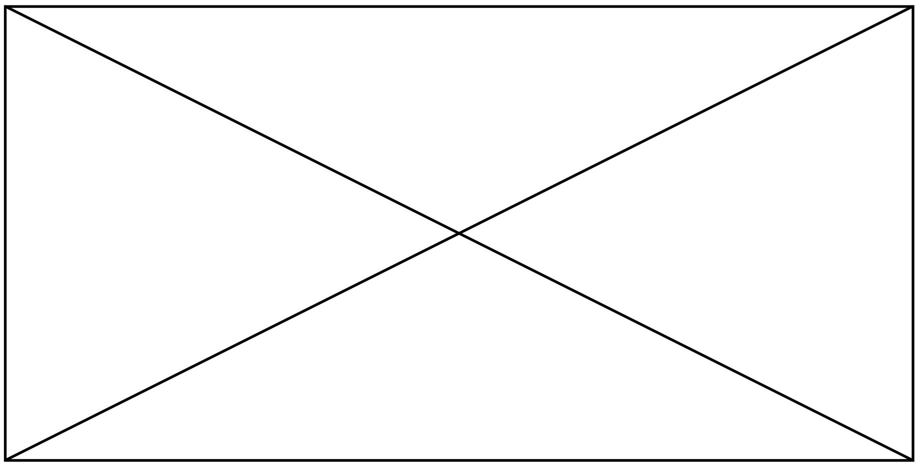

FIGURE 1.3.11. *[1Q]* Some examples of equal graphs.

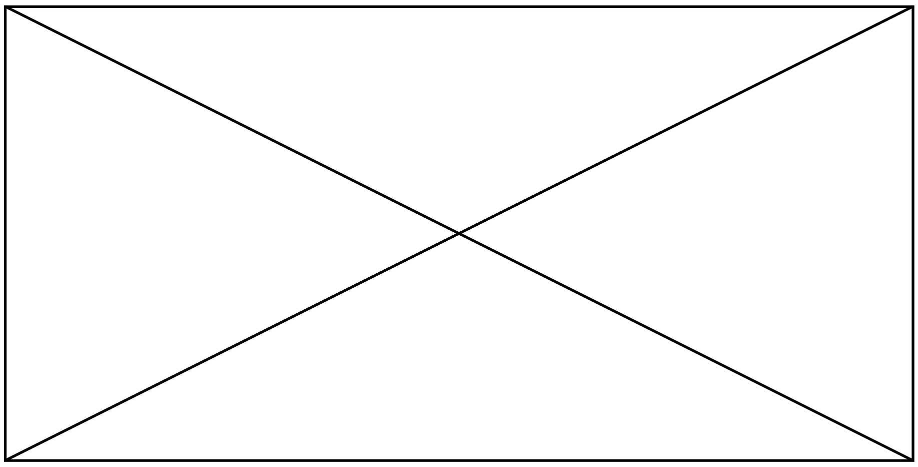

FIGURE 1.3.12. *[1R]* Some examples of unequal graphs.

- each free port in one graph corresponds to a free port in the other graph with the same name;
- the two ports of each wire in one graph correspond to the two ports of a wire in the other graph;
- the ports of each node in one graph correspond to the ports of a node in the other graph with the same symbol.

Figure *1.3.11* shows some equal graphs, and figure *1.3.12* shows some unequal graphs.

1.3.1.10. *Cut.* A *cut* is an untyped graph of a special form. It must contain a constructor node, a destructor node, and no other nodes or wires. The principal port of the constructor must be joined to one of the principal ports of the destructor, and all other ports must be free. Examples of cuts are shown in figure *1.3.13* and figure *1.3.14* shows some untyped graphs that are not cuts.

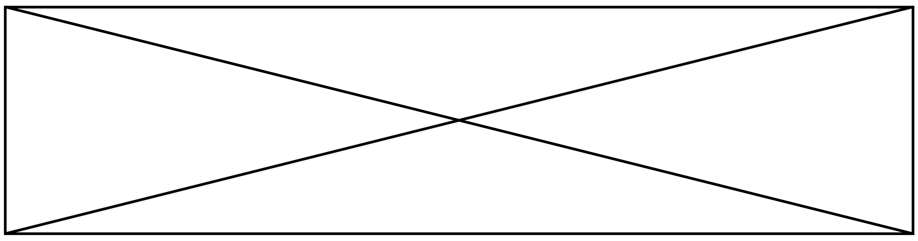

FIGURE 1.3.13. *[1U]* Some examples of cuts.

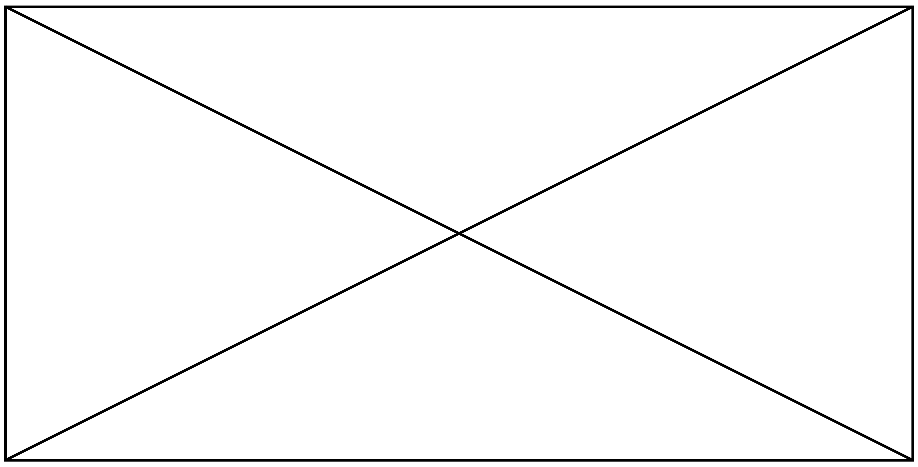

FIGURE 1.3.14. *[1V]* Some untyped graphs that are not cuts. (a), (b) There must be exactly two nodes. (c), (d) One node must be a constructor and the other a destructor. (e) There must be no wires. (f), (g), (h) There must be exactly one junction. (i) The junction must join the principal port of the constructor to a principal port of the destructor.

1.3.1.11. *Type identifier.* A *type identifier* is a string such as 'boolean', 'integer' or 'unit'. Sometimes I use punctuation characters, as in 'list[!integer]'. Although this suggests that the type is constructed in some automatic way from the type 'integer', it is only a suggestion, and in fact 'list[!integer]' is just another type identifier. Type identifiers are equal only if they are identical.

1.3.1.12. *Type.* A *type* represents the state of one of the parties in a two-party communication protocol. It consists of a type identifier and a *polarity*. A *polarity* is either 'in' or 'out'. Given a type identifier t, the input type is written $?t$ and the output type is written $!t$. For example, the type '?list[!unit]' has polarity 'in' and type identifier 'list[!unit]'. Given a type T, its *anti-type* $\overline{T}$ is the type with the same identifier and opposite polarity. Types are equal only if they are identical (but unequal types can each be subtypes of the other, as defined in section 1.3.1.16).

Types often appear as annotations on the ports of a graph, in which case an alternative notation is more appropriate. The polarity is indicated by adding an arrowhead to the port, and the type identifier is written beside it.

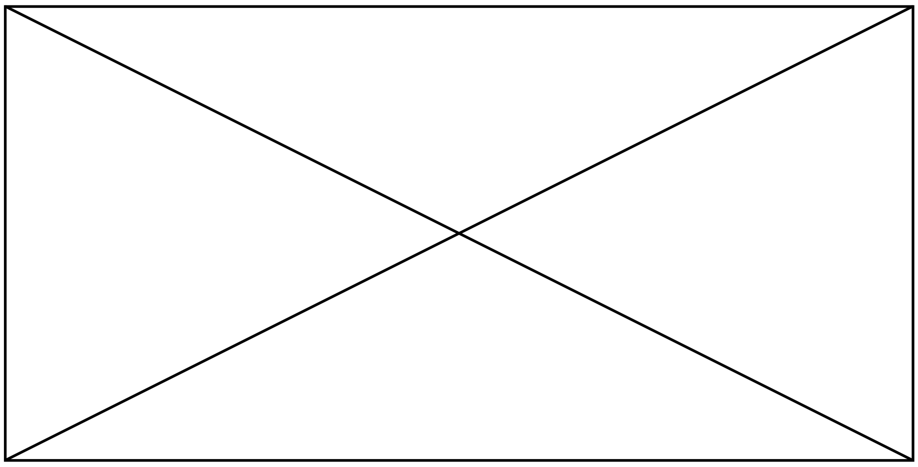

FIGURE 1.3.15. *[1Z]* Some constructor declarations. ‘boolean’, ‘unit’ and ‘natural’ should be familiar. ‘heartbeat’ is a simple protocol for checking that a program has not crashed. The pattern I have obeyed in defining the ‘option’, ‘list’ and ‘stream’ types should be obvious.

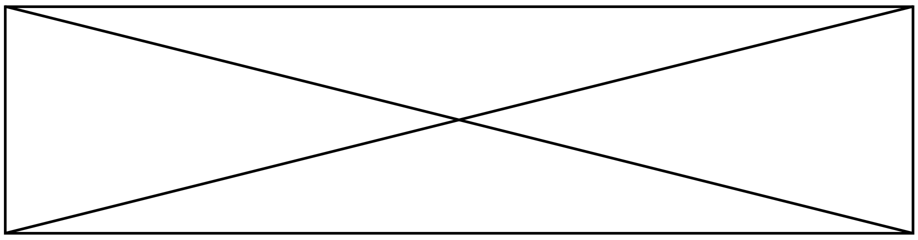

FIGURE 1.3.16. *[1AA]* Some examples of destructor declarations.

1.3.1.13. *Constructor declaration.* A *constructor declaration* declares a legal move in a communication protocol. It is drawn as a constructor symbol with all of its ports annotated with types. The principal port must have an output type (it identifies in which state of which protocol the move can be made) and its type identifier may not be ‘zero’. There is no constraint on the types of the auxiliary ports. Figure *1.3.15* shows some legal constructor declarations.

1.3.1.14. *Destructor declaration.* A *destructor declaration* declares that a destructor communicates in a particular way. It is drawn as a destructor symbol with all of its ports annotated with types. Every principal port must have an input type, but there is no constraint on the types of the auxiliary ports. Figure *1.3.16* shows some legal destructor declarations.

1.3.1.15. *Node declaration.* A *node declaration* is a constructor declaration or a destructor declaration. Halt symbols automatically obey any combination of types, and do not need to be declared. Node declarations are equal only if they are identical.

1.3.1.16. *Subtype declaration.* A *subtype declaration* declares that one communication protocol is contained in another, in the sense that a party using the one is compatible with a party using the other, although the conversation might not explore all of the possible variations. It is written !t <:!u (or, equivalently, ?u <:?t if you prefer), where t and u are type identifiers. A declaration of the form !t <:?u or ?t <:!u is not allowed. The subtype declarations T <: U and $\overline{U}$ <: $\overline{T}$ are equal, but different from all other subtype declarations.

A subtype declaration !t <:!u (or ?u <:?t) is *type-correct* for a set of constructor declarations iff for every declaration of a constructor symbol c with n auximilary ports and a principal port of type !t there is also a declaration of c with n auxiliary ports and a principal port of type !u such that each auxiliary port which in the first declaration has type T has in the second declaration a type U such that U <: T (or $\overline{T}$ <: $\overline{U}$) is another subtype declaration.

Note that a subtype declaration does not *make* the two types compatible; the compiler could easily work out for itself whether one communication protocol contains the other, by trying to find a set of subtype declarations that satisfy the above definition. The declaration simply declares that the compatability is *intended* by the programmer. Subtype declarations make programs more readable and, by giving the compiler some redundant information to check, more maintainable.

Here is a complete list of all subtype declarations which are type-correct for the constructor declarations in figure 1.3.15. There should be enough examples here for you to get a feel for what is going on:

!unit	<:	!option[!bool]	?option[!bool]	<:	?unit
!unit	<:	!option[!unit]	?option[!unit]	<:	?unit
!unit	<:	!list[!bool]	?list[!bool]	<:	?unit
!unit	<:	!list[!unit]	?list[!unit]	<:	?unit
!unit	<:	!list[?unit]	?list[?unit]	<:	?unit
!unit	<:	!list[!list[!unit]]	?list[!list[!unit]]	<:	?unit
!list[!unit]	<:	!list[!list[!unit]]	?list[!list[!unit]]	<:	?list[!unit]
!unit	<:	!natural	?natural	<:	?unit
!option[!unit]	<:	!natural	?natural	<:	?option[!unit]
!stream[!bool]	<:	!list[!bool]	?list[!bool]	<:	?stream[!bool]
!stream[!unit]	<:	!list[!unit]	?list[!unit]	<:	?stream[!unit]
!stream[!unit]	<:	!list[!list[!unit]]	?list[!list[!unit]]	<:	?stream[!unit]

1.3.1.17. *Typed graph.* A *typed graph* represents the run-time state of an entire program. It is formed by annotating every free port of an untyped graph with a type. Thus, each port of a typed graph has both a name and a type; to make it clear which is which, they are drawn as a single annotation of the form 'name:type'. Typed graphs are equal if their untyped graphs are equal, and if corresponding free ports are annotated with the same type.

A typed graph is *type-correct* for a set of node declarations and subtype declarations iff, after exploiting the ambiguity in reading the wires, it is possible to annotate every bound port (in addition to the free ports) with a type, such that:

- every node is surrounded by types exactly as in one of the declarations of its symbol,
- the two ports of every wire are annotated with anti-types, and
- whenever a port of type T is joined to a port of type U there is a subtype declaration T <: $\overline{U}$ (or U <: $\overline{T}$).

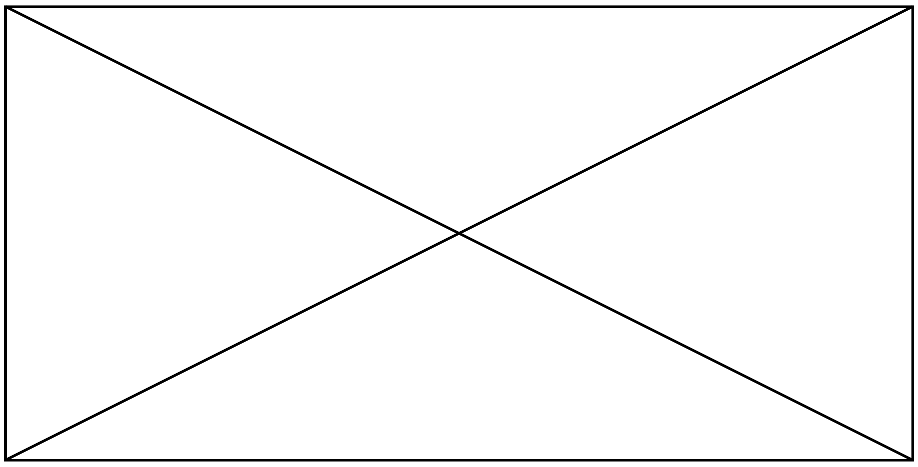

FIGURE 1.3.17. *[1W]* Some examples of rewrite rules.

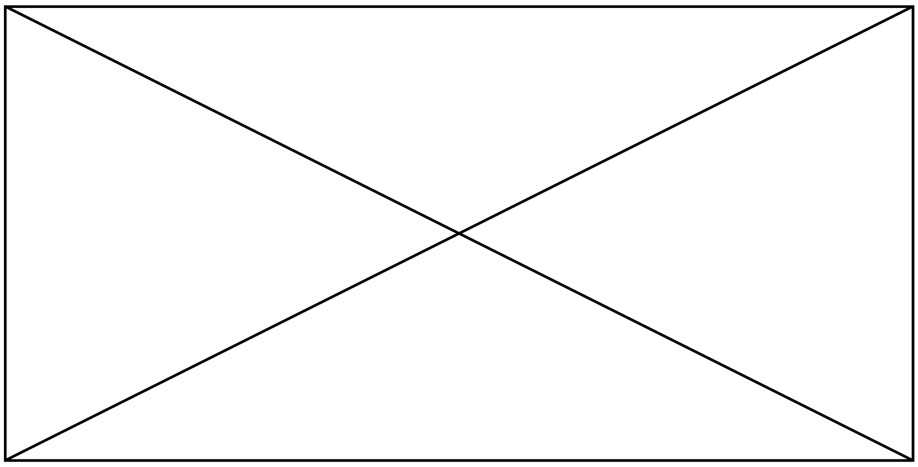

FIGURE 1.3.18. *[1X]* Some examples of things that are not rewrite rules. (a), (b) The redex must be a cut. (c), (d) The redex and reduct must have the same free names.

1.3.1.18. *Rewrite rule.* A *rewrite rule* is a template for reaction steps. It is drawn as a pair of (note) untyped graphs related by an arrow. The graph at the tail of the arrow is called the *redex* and must be a cut. The graph at the head of the arrow is called the *reduct*, and can be an arbitrary untyped graph, except that it must have exactly the same free names as the redex. Figure 1.3.17 shows some examples of rewrite rules, and figure 1.3.18 shows some that are not allowed.

A rewrite rule is *type-correct* for a set of node declarations and subtype declarations iff every assignment of types to free ports that makes the redex a type-correct typed graph also makes the reduct type-correct. The example rewrite rules in figure 1.3.17 are type correct for the node declarations in figures 1.3.15 and 1.3.16 without needing any subtype declarations.

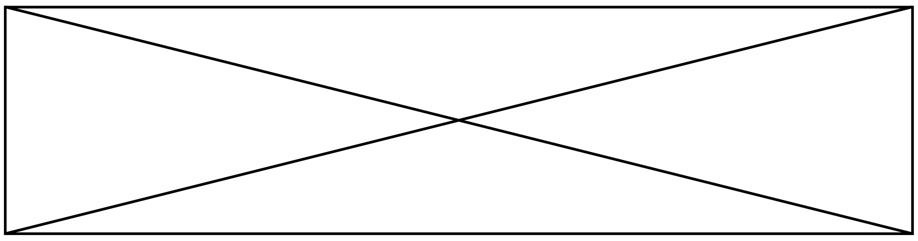

FIGURE 1.3.19. *[1Y]* Names can be omitted from rewrite rules without risk of ambiguity.

I invariably use some notational sugar in drawing rewrite rules. The purpose of the free names in a rewrite rule is to establish an unambiguous bijective correspondence between the free ports of the redex and the free ports of the reduct. This correspondence can be established just as effectively 'by eye', by spacing the free ports around the edge of the graphs, and adopting the convention that (for example) the top-left port of the redex corresponds to the top-left port of the reduct. The names can then be omitted. Figure 1.3.19 illustrates this notational convention.

I also sometimes find it helpful to add arrowheads to the otherwise untyped graphs of rewrite rules. I do this in order to suggest ways in which types might be added to the graphs, in order to help the reader understand the program. The arrowheads have the status of program comments, and have no formal significance. In particular it is still necessary to check every assignment of types to free ports (not just those that are compatible with the arrowheads) in order to determine whether a rewrite rule is type-correct.

1.3.1.19. *Program.* A *program* is the unit of compilation. It consists of a set of node declarations, a set of subtype declarations, a set of rewrite rules and a typed graph called the *main graph*. All node symbols, apart from halt symbols, mentioned in any of the rewrite rules, or in the main graph, must also have at least one declaration. All type identifiers, apart from the special type 'zero', mentioned in any of the node declarations or subtype declarations, or in the main graph, must appear on the principal port of at least one constructor declaration. Programs are equal only if all of their components are equal. Programs are type-correct only if all of their components are type-correct. Examples of type-correct programs are shown in figures 1.2.1, 1.2.4, 1.2.5, 1.5.1 and 1.6.1.

1.3.2. Reaction relation.

The *reaction relation* is a relation on programs which formalises the rewriting behaviour of MIN programs. If it relates two programs P and Q, we write $P \longrightarrow Q$, and say 'P rewrites to Q in a single step'. It forms half of the definition of the MIN's execution model, the other half being the operation of linking two programs.

The intuitive picture of the reaction relation is very appealing. If you can rearrange the main graph of a program P on the page, and then cut out (with a pair of scissors) an untyped graph that looks exactly the same as the left-hand side of a rewrite rule in P, then by replacing it with the right-hand side (which will certainly fit in the hole and remain type-correct) you obtain a program Q such that $P \longrightarrow Q$.

However, this intutive picture deserves some formalisation. It relies on understanding that if you cut through the junction of two ports then they become separated in such a way that they can later be joined to something else. Also, it relies on understanding that if you cut across the middle of a wire then it conveniently breaks into two shorter wires, and that you are not allowed to cut through a node or a free port. You may be willing to agree already that there is only one interpretation that makes sense, but let me continue just in case.

For all its inexactitude, the scissors and glue model correctly captures two useful properties:

- The reaction relation has the property that if $P \longrightarrow Q$ then P and Q have the same node declarations, subtype declarations and rewrite rules, and their main graphs have the same free names, and corresponding free ports have the same type. The differences between P and Q are confined to the untyped part of the main graph.
- The reaction relation ignores all type information. If $P \longrightarrow Q$ and P' differs from P in exactly the same way that Q'differs from Q, and if the difference is confined to the declarations and the types of the free ports of the main graph, then $P' \longrightarrow Q'$.

Thus, if we define the reaction relation for an untyped version of MIN, in which programs have no declarations or types, and consist only of rewrite rules and an untyped main graph, then the definition can immediately be extended to the full typed language. The rest of this section therefore concentrates on the untyped language.

1.3.2.1. *Contexts.* The simplest way to formalise the reaction relation is to think of it the other way up: if a program P has a rewrite $p \longrightarrow q$ and if the main graph of P is $C[p]$ for some graphical context $C[\,]$, then $P \longrightarrow Q$ where Q differs from P only in that its main graph is $C[q]$. It remains only to define the possible contexts for untyped graphs.

A context is an operation which, when applied to an untyped graph, returns a larger[4] untyped graph which contains the original. Roughly, it represents a graph with a hole in it that is left after part of it has been removed with a pair of scissors. More formally:

- Given two untyped graphs p and q, with no free names in common, we can form a new, larger graph by drawing p and q beside each other, without any new junctions. By fixing q, we can regard this operation as a context applied to p.
- Given an untyped graph p with at least two free ports named x and y, we can remove the names and join the two ports together to obtain a new untyped graph. By fixing x and y, we can regard this operation as a context applied to p.
- Any sequence of contexts is a context.

Thus, a typical context might be 'join x to y, draw beside this other graph q, then join u to z and v to w'. Note that contexts are partial functions: the previous example is defined only on graphs that have free names x and y but no other names in common with q (among other conditions). I hope it is clear that these operations correctly and completely capture the intuitive idea of placing one graph inside another.

[4]The resulting graph is larger in the sense that it will contain at least as many nodes and loops, but it might actually have fewer free ports, and contain fewer wires, since it might join two wires together.

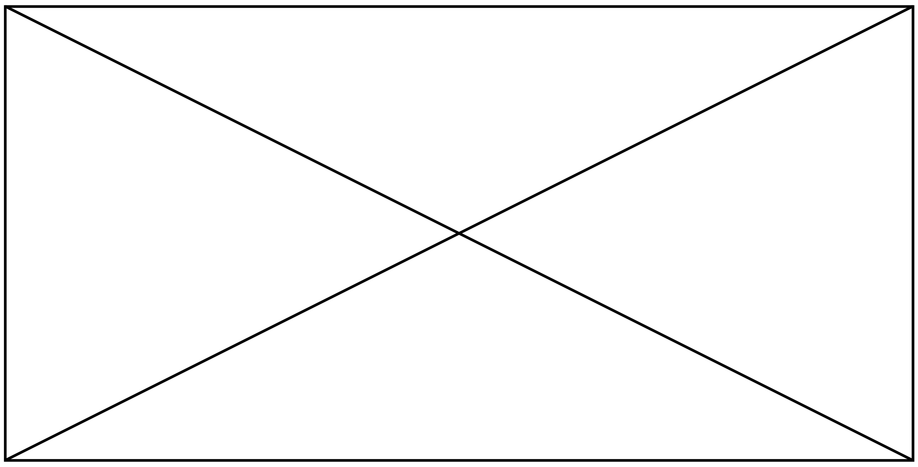

FIGURE 1.3.20. Two equivalent rewrite rules. (b) can be derived from (a) by placing it in a context as shown in (c). Similarly, (d) shows how to derive (a) from (b).

1.3.2.2. *Renamings are contexts.* In section 1.3.1.18 I asserted that the names used in a rewrite rule were important only in that they establish a correspondence between the ports of the left- and right-hand sides, and that any other choice of names that establishes the same correspondence is just as good. I am now in a position to justify this assertion.

Figure 1.3.20 shows two rewrite rules that differ only in that the first uses the names a and b and the second uses the names x and y. By placing both sides of the first rule in the context 'draw a wire with free names u and x and another with free names v and y and join a to u and b to v' we can derive the second rule. Similarly, we can place the second rule in a context to derive the first.

Thus, any reaction that can be derived from one of these rules can be derived from the other. It does not matter which we include in a program. I hope it is clear that this construction can be generalised to any injective renaming of any number of ports.

1.3.3. Linking. The operation of linking two programs forms the second half of the definition of MIN's execution model. A programmer would link programs both to construct a large program from its separately compiled pieces, and also then to launch the resulting program, by linking it to the (running) operating system.

The first step in linking two programs is to ensure that they do not have any destructor symbols or type identifiers in common. Often, this will be true just by coincidence. When it isn't, it suffices to apply a systematic renaming to one or both of the programs. For example, one possible strategy would be to add '1' to the end of all destructor symbols and type identifiers in the first program, and to add '2' in the second. Constructor symbols and names must not be changed.

Next, calculate the set of node declarations, the set of subtype declarations and the set of rewrite rules of the new program by forming the unions of the corresponding sets of the two input programs. The node declarations and rewrite rules of the new program are now finished, but subsequent steps will add some more subtype declarations.

To calculate the main graph of the new program, draw the main graphs of the two input programs beside each other, with no junctions. This is not yet a graph, because any names mentioned in both graphs will now be mentioned twice. For each port of type T with the same name as another of type U, remove all annotations from the two ports, join them together, and add a new subtype declaration $T <: \overline{U}$ (or $U <: \overline{T}$).

Finally, check that all of the new subtype declarations are type-correct. Referring to the definition in section 1.3.1.16, there are three ways a subtype declaration can fail to be type-correct. First, another subtype declaration may be missing, in which case add it and check again. Second, a constructor declaration may be missing, and third, the program may require a subtype declaration $T <: U$ in which T and U have opposite polarities, in which cases give up trying to link the two programs and report an error. These are the only ways in which linking can fail.

The result is a type-correct program. That the subtype declarations are type-correct is true by construction. That the main graph is type-correct follows from the fact that the main graphs of the two original programs were type-correct, since we have carefully added a new subtype declaration for each of the new junctions. That the rewrite rules are type-correct is not at all obvious, since there are now many more ways to type their redexes. However, the relationship of the subtype declarations to the constructor declarations is designed to ensure that there are just as many new ways to type the reducts. It all works out nicely, as proved in chapter 4.

Note that MIN's linking operation is commutative, which is not true in many common programming languages. Note also that MIN's linking operation is often associative, but that associativity breaks down in the case of three programs which all have a name in common.

1.3.4. Implementation. An implementation of MIN is allowed to treat the execution model described in the previous two sections as an algorithm, to be followed to the letter, but this is unlikely to be an efficient way of running MIN programs. MIN's definition therefore grants some freedom to be ingenious. An implementation may modify or even discard parts of a program before executing it, if it thinks that to do so would be a valid optimisation. It may also translate parts of a program into some other more appropriate language, provided it correctly marshals all data that crosses the boundary.

This section defines the limits of what an implementation may do. For the purposes of the definition, I suppose that the implementation works by replacing a slow MIN program with a faster one. This supposition does not limit the applicability of the definition, however. It will work just as well for an implementation that translates into another language, provided the other language has a reaction relation and a linking operation, and can emulate the test required by section 1.3.4.4.

Every implementation must provide a relation $\underline{S}$ on MIN programs with the interpretation that $P \ \underline{S}\ Q$ iff it would be valid to implement P as Q. The relation need not be a function or the inverse of a function; it is permissible to implement different programs in the same way, or to implement the same program in multiple different ways. The relation may also include pairs that the implementation has no intention of using. Typically, $\underline{S}$ will be reflexive, transitive and not symmetrical, but none of these properties is required.

The definition takes the form of four conditions that $\underline{S}$ must obey.

1.3.4.1. *Reduction closure.* The first condition states that if P can be implemented as Q and there is a sequence of reactions from Q to Q' then there must be a corresponding sequence of reactions from P to some P' such that P' can be implemented as Q':

$$\underline{S} \longrightarrow^* \quad \subseteq \quad \longrightarrow^* \underline{S}$$

There is a straightforward intuition behind this condition: the implementation may not follow any execution paths that do not correspond to possible execution paths of the program it is supposed to be running.

1.3.4.2. *Coupling.* The second condition states that if P can be implemented as Q then there is a sequence of reactions from P to some P' such that Q can be implemented as P':

$$\underline{S} \quad \subseteq \quad \longrightarrow^* \overset{-1}{\underline{S}}$$

The purpose of this condition is to ensure that the implementation follows at least one of the execution paths of the program it is supposed to be running, but it is certainly stronger than that. Perhaps I will weaken it in the future. It is good enough for now.

1.3.4.3. *Context closure.* The third condition states that if P can be implemented as Q then for any type-compatible program R the result of linking P and R can be implemented as the result of linking Q and R. Writing $\times$ for the operation of linking two programs:

$$P \;\underline{S}\; Q \quad \Longrightarrow \quad P \times R \;\underline{S}\; Q \times R$$

This condition also has a straightforward intuition: if the implementation considers it valid to implement P as Q, it must not change its mind on discovering that it will be linked with R.

Note that it is not necessary for the implementation to consider the result of linking Q and R to be a *good* way of implementing the result of linking P and R, only that it is a *valid* way. It might be able to do something ingenious to R, and perhaps something even more ingenious after linking it with Q.

1.3.4.4. *Soundness.* To state the fourth condition it is necessary to choose a constructor symbol which is very unlikely ever to appear in a real program. For example, its symbol might include a large randomly generated number that is kept secret from programmers. Let us call this symbol 'test'.

The fourth condition states that if P can be implemented as Q and the main graph of P includes a node with symbol 'test', then the main graph of Q must also include such a node, and *vice versa*. Writing test(P) for the predicate 'P contains a 'test' node':

$$P \;\underline{S}\; Q \quad \Longrightarrow \quad \text{test}(P) = \text{test}(Q)$$

The purpose of this condition is to ensure that Q bears at least some resemblence to P. Without this condition, it would be perfectly valid to implement any program as any other program. With it, P and Q must be sufficiently similar that no program linked with one can tell (in the sense that it can use the information to decide whether or not to produce a 'test' node) that it is not linked with the other.

The condition is even stronger when applied to programs which, however rarely, sometimes produce a node with symbol 'test'. Optimisation of such programs will be impaired. However, we have specifically chosen 'test' so that we can expect never to encounter such programs in practice.

1.4. Graphical Notation

The use of a graphical notation for MIN is not necessary for any semantic or mathematical purposes. On the contrary, it causes some complications, because the structures that are most natural graphically are not the same as the ones that are most natural textually. There were several opportunities to simplify the textual notation at the price of damaging its interpretation as a description of a graph, but I resisted them all.

Nor is the graphical notation intended to make it any easier to program in MIN. The ability of programmers and their tools to construct extremely complicated structures out of unimaginably many simple pieces already greatly outstrips our ability to understand and manipulate the results. On the contrary, MIN imposes extra discipline on the simple pieces and on the way in which they are joined together.

The purpose of the graphical notation is to make it easier to understand what a MIN program says. It is particularly good at drawing attention away from semantically irrelevent information. Principally, this is exceedingly helpful when manipulating programs algebraically, but it has numerous other advantages:

- Despite its expressive power, MIN is a language that can easily be taught to any programmer, even though they may have no understanding of the basic tools of discrete mathematics, such as grammars and sets, nor of the lower levels of computer architecture, such as pointers and threads.
- A graphical model of computation is a helpful intuitive crutch on which to base an implementation. Its mapping onto the pointer structures used to represent a running program in memory is much more direct than is possible from a term rewriting model.
- Even when using MIN's textual notation for a mathematical proof, the possibility at any moment of interpreting a term as a graph is exceedingly useful. It guides the reasoning down productive paths, and it promptly draws attention to mistakes.

1.4.1. Connections. One of the ways in which a graphical notation clarifies the structure of a program is that parts of a graph that are not connected to each other can never affect each other's behaviour. Moreover, parts of a graph that are distantly connected cannot affect each other for some time. We might say that the speed of light in the graph is one node per rewrite.

In contrast, in a textual program, both control and data can jump all over the code, following the names of variables and subroutines, and the grammar of loops and conditional structures. Doubtless, there *is* a structure that the influences are obeying, but it bears little resemblance to the superficial form of the program. In fact, it is some sort of graph.

Explicity locality of interaction is helpful in a number of ways. It is clear that any fragment of a program can be studied by considering only its interior and the information that crosses its boundary. This immediately tells us to put types at the boundary only, for example. The idea also enables a manageable test for program equivalence. The simplest possible definition of 'interior' and 'boundary' is obtained from the simplest possible definition of 'local', and that is what a graphical notation provides. The popular childrens' puzzle demonstrates that any four-year-old can trace a wire from one end to the other.

1.4.2. Structural congruence. A program in any language contains information that does not affect its behaviour. For example, in a textual program, independent subroutines and variables can be declared in any order; the order is preserved in the code, but has no semantic effect. Also, programs formed by linking smaller programs often retain artifacts of the way in which they were organised prior to linking, although their organisation has no run-time effect. There are also much more subtle examples: the difference between two stable sort algorithms is redundant information. In general, it is undecidable whether the difference between two programs is redundant.

Programs in both graphical and textual notations have this redundant information, but a graphical program has much less. Arguably, the defining feature of an unambiguous textual grammar is that every term retains perfect information about the pieces from which it is made, and the operations used to join them together. In contrast, there are many ways in which a given graph can be broken into two pieces. Once the pieces are joined, it is impossible to tell what they were; that redundant information is simply forgotten.

Redundant information in a program is interesting because it is only the redundant information that can be changed in order to optimise a program. By using a graphical notation, a swathe of optimisations is rendered unnecessary (automatic, trivial), because they change only the information that is forgotten anyway. Moreover, it is common for one optimisation to reveal another; if the first is unnecessary in a graphical notation, it becomes that much easier to find the second. This is particularly advantageous in cases where the first optimisation on its own is not obviously beneficial.

The precise extent of the information that is retained by MIN's textual notation but forgotten by its graphical notation is one of the concepts that I formalise in chapter 3. I define an equivalence relation called the *structural congruence*, which relates any two textual programs that correspond to the same graphical program.

1.5. Linearity

My decision to use a linear language is perhaps one of the most controversial in this thesis. This section is an attempt to explain it. In order to avoid discussing two unfamiliar concepts at once, I will start by discussing linearity in the context of an ordinary high-level language, and then go on to the implications for MIN.

1.5.1. Programming with linearity. There are many good reasons to want a high-level programming language (by which I do not mean MIN) to be able to express that some values are *linear*, and to be able to check and enforce that contract. By this I mean that such a value cannot be copied or discarded. It should be a static, compile-time error to try. For example:

- If you read from a variable that holds a linear value, for example to pass it to a subroutine, then the variable immediately goes out of scope, so you can't read from it again. If you did, you would have implicitly copied the value.

- If a subroutine returns a linear value, you must store it somewhere, otherwise you would have implicitly discarded it.
- You can't assign to a variable that already has a linear value, as this would implicitly discard the old value.
- No linear value may be in scope at the end of a subroutine (after the 'return' statement), because values which 'drop off the stack' are implicitly discarded.

I find it surprising that almost no programming languages in common use can do these checks, because they are simple and would catch many common bugs. For example:

- A block of memory allocated with 'malloc()' should be a linear value. If it were, it would be impossible to forget to free memory, or to free it twice. Also, it would be impossible not to initialise a variable that is intended to point to an allocated block.
- Any program that disables interrupts should receive a dummy linear value representing the responsibility to re-enable them.
- A network connection, once opened, should be a linear value. This would ensure that there is always a unique part of the program responsible for it, thereby ensuring that all connections are eventually closed, and that no more than one thread at a time races to send data.
- A file handle should be a linear value, to ensure that it is eventually closed.

In summary, linearity is a recurring idiom that often appears in the human-readable documentation for a program, but that would be much better expressed formally in the language itself, perhaps in the type scheme, so as to enable the compiler to check it. Programmers already have to think linearly: why can't they write down their thoughts?

1.5.2. Compiling linearity. We are all familiar with the advantages of a pure functional language, most of which are also shared by the functional subset of an impure language. Perhaps the most important of these advantages is a high-level one: when using other people's code it is helpful not to have to understand whether values are shared or not. In addition, at a low level, a functional language presents significant opportunities to its compiler. For example:

- having read a value from a functional data structure, it is not necessary ever to read it again, as it is guaranteed to be the same each time it is read. Conversely, it does no harm to read a value more than once. The decision can be made by the compiler on grounds of efficiency.
- We can safely compile a nested tuple $(a, (b, c))$ as a triple (a, b, c). As long as every part of the program adopts the same policy, the semantics of the program will not change. It does not matter whether sub-tuple (b, c) is shared or not, as nobody can modify it. This will improve locality, and will probably reduce memory consumption and access time, and also garbage collection time.
- We can apply techniques such as common sub-expression elimination to structures as well as to primitive values, without worrying about whether they are shared.
- A generational garbage collector can be simplified and optimised using the assumption that all values are functional.

All these benefits are also benefits of linear values. A linear contract is strictly stronger than a functional contract: better than not needing to worry if a value is shared is to know that it is not shared. In addition to the benefits of functional programming, linear programming offers more. For example:

- When a linear structure is deconstructed, we can immediately free and reuse the memory it consumed, since we know that it was not shared. This reduces memory consumption and lightens the load on the garbage collector.
- We can perform destructive update: we can overwrite a field of a linear structure in the certainty that nobody but ourselves will notice.
- Given two linear pointers to arrays, we can be sure that they point to different arrays, so there are no aliasing problems. Loads and stores to one array can be re-ordered freely with loads and stores to the other array.

In summary, it is typically possible to compile linear values more efficiently than functional values, which in turn can be compiled more efficiently than values for which the programmer has not provided any useful contract.

1.5.3. Input and Output. Linear values can be used to construct a good model of the real world. One way to understand this is to think of IO as an in-place destructive update operation applied to the world. This picture lets us use the same intuitions for IO as for normal programming. For example, we immediately see that the world cannot be modelled as a first-class value in a pure functional language. It simply does not obey the contract. We also see that we are going to have to be careful if ever we want more than one program to perform IO at the same time. We will have to wrap the world in an arbiter of some sort, and share only the arbiter, not the world itself.

Take a sound card, for example. To play a sound, we have to choose a sample rate (e.g. 44.1kHz) and a sample format (e.g. signed 16-bit stereo) and then send a stream of data to the sound card. If two programs were to do this simultaneously, nothing good would result. The data from the two sources might be interleaved, causing the sound to chop unpredictably from one to the other. If the sample rates are different, one of the sources would be played at the wrong pitch. If the formats were different, one would be misunderstood completely. It would be a fiasco. Furthermore, this example has used quite a high-level model of a sound card. If we go a little lower, and try to install two drivers for the same sound card, we'll probably hang the machine. To avoid these problems, we should wrap our sound card in a mixer, and expose only the mixer as the high-level interface. We can copy references to the mixer, and this allows the sound card to be shared, in an informal sense. The mixer is acting as an arbiter. However, mixing sound sources requires a certain amount of arithmetic, especially if they use different sample rates, so the mixer clearly has to be a program. Nobody is going to confuse it for a fundamental computation primitive. The presence of a mixer in our operating system therefore does not answer the question of how to model the sound card, because the mixer itself needs such a model. What should that model be? We clearly must have a model, even if it is unwritten. The answer is that it must be linear. Moreover, we cannot sit back contented that the linearity has been buried behind the abstraction layer provided by the mixer. Although we can open many connections to the mixer, each of those connections is a different object with a distinct identity, and suffers from all of the same problems as the physical sound card. We cannot get away from the linearity.

A similar argument can be made for other real-world objects. A disk is an interesting example: although in principal it is a closed system which does not communicate with the outside world, we still cannot share it in a functional way, because that might involve copying its contents, and it is just too big. The other half of the argument (that the disk must be hidden behind an arbiter which treats it as a linear object) is largely unchanged.

In summary, it is not possible to avoid linearity in a programming language, even if we wanted to, if we also entertain ambitions to perform IO. If we fail to model it in the language, it will simply crop up in the documentation. On the bright side, IO provides a rich collection of examples of linear contracts.

1.5.4. The return stack. I have said that most languages cannot enforce linear contracts on first class values. That certainly does not mean that these languages do not enforce any linearity at all. Any language that supports subroutines, be they pure functions or not, compiles into machine code that includes a pervasive data structure with a strict linear contract: the return stack. This serves as an excellent example of linearity buried in superficially non-linear languages. It certainly isn't the only example.

A typical calling convention might include the following contract for the return stack. The stack pointer points to the middle of a block of memory called the stack. Your subroutine must preserve the value of the stack pointer. The word to which the stack pointer points and all words later in the block must be treated as opaque data; you may not rely on their contents, nor may you modify them. The words earlier in the block, in contrast, are free for you to use in any way that you like. In particular, if you decrease the stack pointer, call a subroutine, and then restore the stack pointer to its original value, and if the subroutine you call obeys this calling convention, then you will also obey this calling convention, and furthermore the words between the two values of the stack pointer will not be modified. This provides a cheap way of allocating storage.

What is going on here is that the part of the stack above the stack pointer is a value which you receive, which you cannot discard or copy, and which you must return. In other words, it is linear. It is also opaque, but that is beside the point. The right to modify the free space below the stack pointer is another linear value. Decreasing the stack pointer is a destructive operation on the free space, which splits it into a useful block of memory (also linear) and a smaller amount of free space. Increasing the stack pointer is another destructive operation which reverses the split. Everything on the stack is completely linear. It's an error to allocate a block on the stack and then return a pointer to it, because you must not share a value that other parts of the program assume is linear.

If you look closely at the above contract, you will see that it explicitly uses two of the three optimisations I described in section 1.5.2 which I said could be applied only to linear values: storage is freed and can be reused immediately, and destructive update is the rule. This is most of what makes stack allocation more efficient that heap allocation. Readers familiar with approaches to aliasing problems will recognise the third optimisation too: arrays allocated on the stack are known to be different.

At this point I must mention the approach to IO adopted by the programming language Haskell. Haskell is a pure functional language, but it manages to avoid many of the ugly IO problems encountered by other pure functional languages. It does this using a contorted type-checking trick (the 'IO monad' type constructor) which leverages the linearity of the return stack to obtain a sensible abstraction of the world. Roughly, the idea is that a functional program embeds itself in a lazy list of IO operations that it wishes to perform, and then returns it. The uniqueness of the return value of a function is what prevents the world from being copied. This is utterly perverse, but it seems to work. Why does Haskell not simply have first-class linear values? Who knows!

1.5.5. Encodings. So what has all this got to do with MIN? Let me found my argument on my stated motivation: MIN is designed to improve our understanding of the meanings of programs written in multi-threaded languages by encoding them into a form that can be manipulated algebraically. In the preceding sections I have been arguing that almost all programs involve linearity, sometimes expressed only in their documentation, but just as often expressed formally, for example in their use of subroutines. I have also highlighted several ways in which an understanding of the linearity in a program is essential in order to optimise it (manipulate it algebraically). It is therefore highly desirable that MIN be able to express linearity formally.

If MIN were not able to express linearity formally, then it would not help nearly as much in understanding the meanings of programs, because much of that meaning would be lost in the process of encoding them into MIN. Any encoding of one language into another which maps linear values onto non-linear values will be lossy (incomplete), in the sense that it will discard information that would have been useful for validating optimisations. I do not mean to claim that this is the only loss mechanism, or that MIN permits lossless (complete) encodings from all other languages, but I do mean to claim that linearity is one of the more significant issues and that it is worth getting it right.

1.5.6. Sharing. No matter how much linearity there is behind the scenes in real programming languages, the fact remains that first-class values in almost all languages are non-linear and can be shared. Can we take the argument for including linear values in MIN and turn it around in order to argue for the inclusion of non-linear values too? No. The argument breaks down because it turns out to be possible to encode sharing as computation using only the linear primitives that MIN provides.

For example, recall the AND gate program in figure 1.2.1, and suppose we wanted to write a non-linear version of it that can be shared and used repeatedly. Figure 1.5.1 shows how to achieve this. The idea is to write a program that acts as an AND server. The server understands two kinds of request. First, it knows how to reproduce, which allows the population of servers to increase to meet the needs of the clients. Second, it can construct a one-shot AND gate identical to that in figure 1.2.1. This is not a particularly convenient form of sharing (for example, it does not provide a 'discard' operation) but it is simple, it supports all the expected equivalences, and it illustrates the point. A more sophisticated mechanism is used in chapter 6.

However, the sharing question is much more complicated than it looks. There are at least four quite different kinds of sharing in common use in real programming languages. In Java, sharing is many-to-one. In the π-calculus and similar message-passing languages, sharing is many-to-many. In funtional languages, sharing is many-to-one, but with referential transparency as an extra condition. In Prolog, and unification-based type inference algorithms of some languages, a shared variable has quite a different meaning again. To these we might also be tempted to add the sound mixer discussed in section 1.5.3. All of these are recognisably a form of sharing, in that they are implemented using pointers, and that copying is

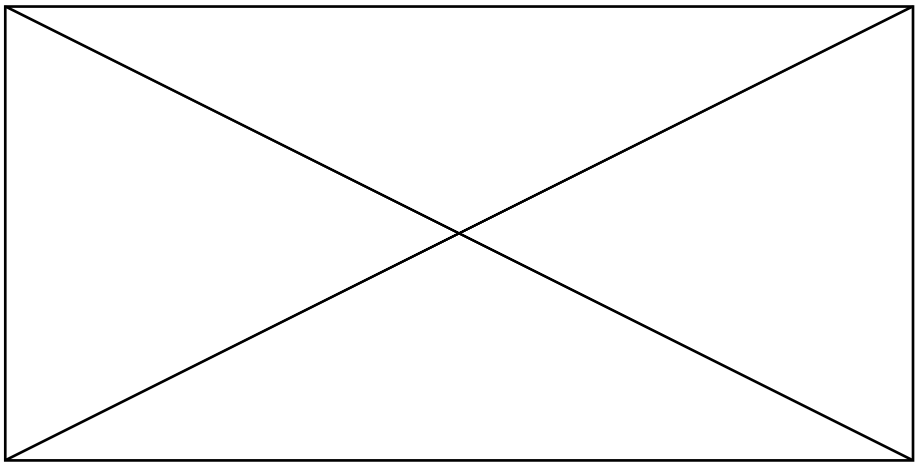

FIGURE 1.5.1. *[1G]* A program that implements a re-usable AND gate.

(the dual of) a commutative, associative operation. However, they have little else in common.

In the face of this variety, the task of adding a sharing primitive to MIN seems somewhat ill-defined. Java-like sharing can already be encoded losslessly in MIN as it stands, as can the sound mixer (given exact arithmetic). There is a variant of the π-calculus whose sharing MIN can encode losslessly, but other features which are difficult to separate from its sharing model preclude a complete encoding of the original π-calculus. Zeroth-order functional sharing can also be encoded losslessly, but incompleteness creeps in for first- or higher-order functional sharing. I don't know how Prolog fares. Certainly any sharing primitive we might wish to add should address a source incompleteness, but it is clear that to address all such sources we will need several new primitives. We might expect to encounter new forms of sharing in the future, requiring yet more special treatment. Somewhere, we have to draw a line.

Moreover, given MIN's capability to encode some sharing already, one could conceive of addressing some of the remaining incompleteness using a primitive that is not a sharing primitive at all. For example, I find it plausible that the gap between Java-like sharing and functional sharing could be filled by a sufficiently powerful type scheme. Such a type scheme might be independently useful, and therefore a more worthy addition to MIN than functional sharing alone. In short, it is not clear that the 'sharing' part of 'functional sharing' is the part that MIN lacks.

In summary, I do not think that our understanding of the meanings of multi-threaded programs will be improved by adding a sharing primitive to MIN at this stage. Certainly we must understand the linear case first, as it is already remarkably complete. Indeed, it remains possible that it will turn out that sharing adds nothing new.

1.6. Program Equivalence

This introduction would not be complete if it did not give some indication of what is involved in proving that two programs are equivalent. This section gives a

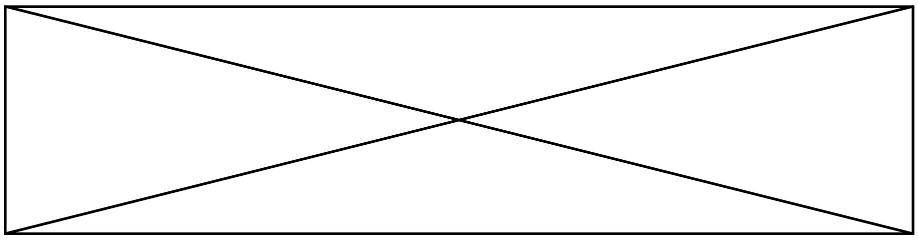

FIGURE 1.6.1. *[1AC]* An implementation of a 'join' concurrency primitive. This program sends a 'nil' message from the free port z only when it has received 'nil' messages at both x and y.

worked example of such a proof. For this purpose I have chosen the simplest non-trivial proposition I could find: that a 'join' concurrency primitive is symmetrical, even though its implementation (shown in figure 1.6.1) is not.

The program is linear, and can only be used once. It has three free ports: two inputs (x and y) and one output (z). All are of type 'unit', which permits just one legal move: the sending of a 'nil' constructor (which I draw as a small empty triangle) with no auxiliary ports. The specification is that once the program has received a 'nil' message at both inputs, but not before, it sends a 'nil' message at its output.

The implementation is asymmetrical because it waits exclusively on one of the inputs, and only after receiving a 'nil' message at that input does it look at the other. Intuitively, it does not matter what the program looks at, only what it sends, and so it should not matter in which order the program examines its inputs. In the language of section 1.3.4, it ought to be valid to implement the program as its mirror image (with the inputs x and y exchanged) and *vice versa*. This indeed turns out to be true.

1.6.1. Labelled transitions. One of the nice things about using a symmetry as an example is that we only need to prove half of the proposition. However, wringing the desired conclusion from the language definition is not a trivial task. The most problematic part is context closure (section 1.3.4.3), which requires us to anticipate all programs with which our example could possibly be linked. A proposition about a trivial little example appears to involve almost the whole language.

The example becomes as trivial as one might hope when the technique of chapter 3 is applied. The technique is based on the idea (mentioned in section 1.4.1) that the behaviour of a program can be completely understood by looking at its interior and at the information that crosses its boundary. The internals of the program are satisfactorily represented by the reaction relation. To these we add *input transitions* and *output transitions*, derived using the rules in figure 1.6.2, which represent respectively the action of receiving and sending a single message. The resulting relation, which represents both internal and external behaviour, is called the *labelled transition relation*.

Thus, there are three kinds of labelled transition:

- A typical input transition is of the form $P \xrightarrow{xc(\vec{y})} Q$, which reads 'program P receives a message at the free port x with constructor symbol c

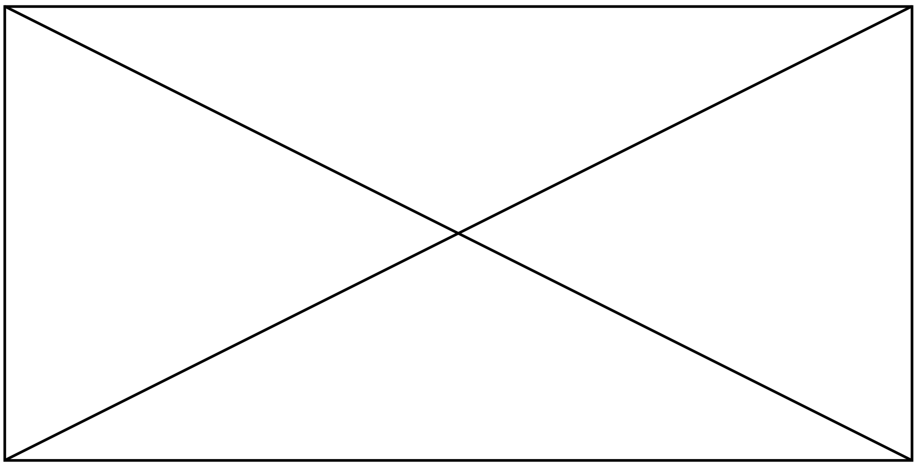

FIGURE 1.6.2. *[1AD]* The rules for deriving input and output transitions. P is a graph, c is a constructor symbol, x is a name an t is a type. $\vec{y} : \vec{u}$ and $\vec{z} : \vec{v}$ represent zero or more names, each with a type. Only the main graphs are shown. Type-checking restrictions apply.

and auxiliary ports $\vec{y}$ and the result is a new program Q'. Note that P must have a free port named x, but no free ports labelled $\vec{y}$, whereas Q no longer has a free port x but acquires free ports $\vec{y}$. The program P has no control over the messages it receives, so the labelled transition system must include all possible input transitions, constrained only by the type-checking rules. The program Q is completely defined by P and the message.

- A typical output transition is of the form $P \xrightarrow{\overline{x}c(\vec{y})} Q$, with a similar interpretation. For outputs, P is completely defined by Q and the message, and no output transition is possible unless P is of the required form. In that sense it has control over what it sends.
- An ordinary reaction $P \longrightarrow Q$ can be written $P \xrightarrow{\tau} Q$ in order to emphasize that it is part of the labelled transition relation.

In the special case of our example program, the only constructor symbol is 'nil', and it never has any auxiliary ports. This keeps the labels simple, and ensures that the part of the labelled transition relation reachable from our starting point is finite. It is shown completely in figure 1.6.3. For a more complicated program, for example one that manipulate lists of arbitrary length, the labelled transition system would be infinite.

1.6.2. Coupled simulation. The following alternative and much more convenient definition of when an implementation is valid is equivalent to that in section 1.3.4 (this theorem is the principal result of chapter 3). As before, the compiler is required to provide a relation $\underline{S}$ with the interpretation that $P \;\underline{S}\; Q$ means that it is valid to implment P as Q, and again the definition takes the form of some conditions that $\underline{S}$ must satisfy. However, the conditions are different, with the result that it is possible to satisfy them using a much smaller relation. We retain reduction closure and coupling, but drop context-closure (which is the one that causes the problems for the old definition) and replace it with a new condition phrased

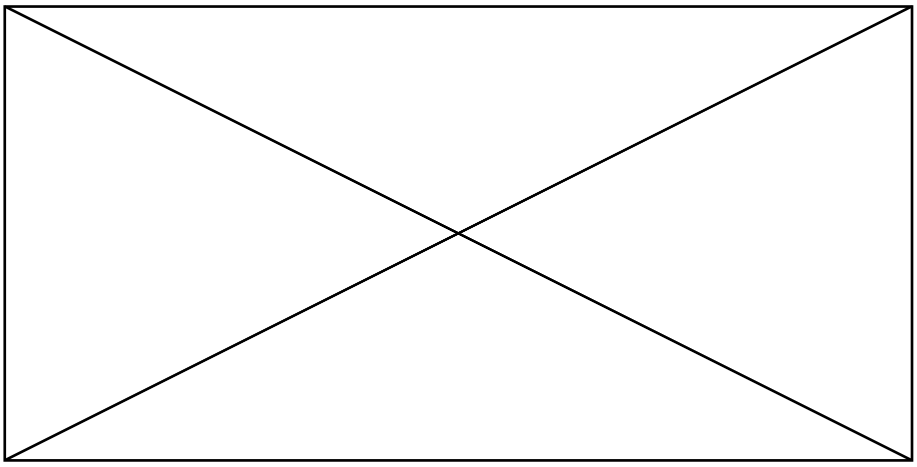

FIGURE 1.6.3. *[1AE]* The part of the labelled transition relation reachable from the program in figure 1.6.1. Only the main graphs are shown.

in terms of labelled transitions. The new condition is strong enough that we no longer need the separate soundness condition. A relation that satisfies the new conditions is called a *coupled simulation*.

DEFINITION 1.6.1. Say a relation $\underline{S}$ on programs is a *coupled simulation* iff it satisfies the following conditions, in which α is an input or an output label:

$$\begin{aligned} \underline{S} \longrightarrow^{*} &\subseteq \longrightarrow^{*} \underline{S} \\ \underline{S} \xrightarrow{\alpha} &\subseteq \longrightarrow^{*} \xrightarrow{\alpha} \longrightarrow^{*} \underline{S} \\ \underline{S} &\subseteq \longrightarrow^{*} \underline{S}^{-1} \end{aligned}$$

This definition has the property that given two coupled simulations, their union is also a coupled simulation. This means that the compiler can provide a different relation $\underline{S}$ for every program it has to compile, safe in the knowledge that the union of all the relations it has ever used will satisfy the language definition. For the purposes of our example, we should therefore feel no shame in producing a tiny relation that is no use for any other example.

Referring to figure [1AE], name the seven programs involved A to G, with A being the initial program, so that the labelled transition system looks like this:

$$\begin{tikzcd} A \arrow[r, "x\text{nil()}"] \arrow[d, "y\text{nil()}"] & B \arrow[r, "\tau"] \arrow[d, "y\text{nil()}"] & C \arrow[d, "y\text{nil()}"] & \\ D \arrow[r, "x\text{nil()}"] & E \arrow[r, "\tau"] & F \arrow[r, "\overline{z}\text{nil()}"] & G \end{tikzcd}$$

Similarly, name the mirror images of these programs A' to G', so that their labelled transition system looks like this:

$$\begin{tikzcd}
A' \arrow[r, "x\mathrm{nil}()"] \arrow[d, "y\mathrm{nil}()"] & D' \arrow[d, "y\mathrm{nil}()"] \\
B' \arrow[r, "x\mathrm{nil}()"] \arrow[d, "\tau"] & E' \arrow[d, "\tau"] \\
C' \arrow[r, "x\mathrm{nil}()"] & F' \arrow[d, "\overline{z}\mathrm{nil}()"] \\
 & G'
\end{tikzcd}$$

Our task is to construct a coupled simulation $\underline{S}$ such that $A \;\underline{S}\; A'$ and $A' \;\underline{S}\; A$. Since we know the answer will be symmetrical, let us adopt the convention that if (for example) $B \;\underline{S}\; D'$ is in the relation then it is understood that $B' \;\underline{S}\; D$ is also in the relation.

Starting from $A \;\underline{S}\; A'$, we can fill up the relation as follows:

- Because $A \;\underline{S}\; A'$ and $A \xrightarrow{y\mathrm{nil}()} B'$ we have to find a program P such that $A \longrightarrow^* \xrightarrow{y\mathrm{nil}()} \longrightarrow^* P$ and $P \;\underline{S}\; B'$. The only candidate for P is D, so we must have $D \;\underline{S}\; B'$.
- Because $D \;\underline{S}\; B'$ and $B' \longrightarrow C'$, and because the only program P such that $D \longrightarrow^* P$ is D itself, we must have $D \;\underline{S}\; C'$.
- Because $A \;\underline{S}\; A'$ and $A' \xrightarrow{x\mathrm{nil}()} D'$ we must have at least one of $B \;\underline{S}\; D'$ and $C \;\underline{S}\; D'$. Let us suppose we have both.
- Because $D \;\underline{S}\; B'$ and $B' \xrightarrow{x\mathrm{nil}()} E'$ we must have at least one of $E \;\underline{S}\; E'$ or $F \;\underline{S}\; E'$. Let us suppose we have both.
- Because $D \;\underline{S}\; C'$ and $C' \xrightarrow{x\mathrm{nil}()} F'$ we must have at least one of $E \;\underline{S}\; F'$ or $F \;\underline{S}\; F'$. Let us suppose we have both.

Note we have now decided that either of E and F may be implemented as either of E' or F'. Intuitively we are saying that (in this example) the $\xrightarrow{\tau}$ steps are deterministic and do not change the meaning of the program. Similarly we have decided that either of B and C may be implemented as D' and that D may be implemented as either of B' or C'. Continue:

- Because $B \;\underline{S}\; D'$ and $D' \xrightarrow{y\mathrm{nil}()} E'$ we must have $E \;\underline{S}\; E'$ or $F \;\underline{S}\; E'$. We already have both. $C \;\underline{S}\; D'$ gives us another reason for the latter.
- Because $E \;\underline{S}\; E'$ and $E' \longrightarrow F'$ we must have $E \;\underline{S}\; F'$ or $F \;\underline{S}\; F'$. We already have both. $F \;\underline{S}\; E'$ gives us another reason for the latter.
- Because $F \;\underline{S}\; F'$ and $F' \xrightarrow{\overline{z}\mathrm{nil}()} G'$ we must have $G \;\underline{S}\; G'$. $E \;\underline{S}\; F'$ gives us another reason.

I have now checked everything except the coupling condition (the third line of definition 1.6.1) but you can check that it is also satisfied. We therefore have a valid coupled simulation, which completes the proof.