CHAPTER 1

Introduction

CHAPTER 2

Background

CHAPTER 3

MIN's Theory

CHAPTER 4

Types

Different programming languages have type schemes for different reasons. Indeed, the word 'type' is widely applied and often misunderstood. I will therefore precede the discussion of MIN's type scheme with a brief review of some of the different kinds of type scheme. *[Is this a good idea?]*

4.1. Type schemes

C provides an extreme example, since its type scheme is not even sound. It is complete, however, so there is some value in checking it: since any legal program can be made type-correct, type errors are indicative of errors. C confuses matters further by mixing up its representation types with its arithmetic operations. For example, casting a 'float' to an 'int' can change its numerical value, without straying outside the language definition (it rounds towards zero). If ever one designed an untyped version of C (or Java for that matter), these so called 'type casts' would have to remain.

For all the advantages of this design, there are good reasons not to design languages like that any more. (People still do, of course.) What, then, is a type scheme? Let us adopt the following informal definition (which excludes what C calls a type scheme):

DEFINITION 4.1.1. (Type scheme)

A type scheme is a statement which, if it is ever true of a program, remains true at all later points in the program's execution.

The property which remains true is called the *subject reduction* property of the type scheme.

4.1.1. Dynamic types. Let me illustrate definition 4.1.1 with another extreme example: Lua (Python is similar). In Lua, there are a finite number of types,[1] including 'table' and 'number'. A value of type 'table' is (a pointer to) a hashtable, whereas a value of type 'number' is a double-precision floating-point number. Although one could contrive pathological examples, it is usually not difficult to distinguish numbers from hashtables, even on 64-bit platforms where each occupies 64 bits, because certain 64-bit values are not legal numbers, and because hashtables have a highly redundant structure and must reside in heap-allocated blocks.

Lua has what is called a *dynamic* type scheme. Every value is stored in memory beside its type, even at run-time. Arithmetic operations operate on the types as

[1] I am ignoring tag methods.

much as on the values. For example, the result of adding two numbers is a number. Indeed, the type can even affect the action of arithmetic operations on the value, since attempting to divide a number by a table produces an error, at run-time, even if the table happens to be a legal number. It is also possible to examine the type of a value using an 'if' statement.

It is nonetheless fair to call this a type scheme, because the following statement is always true: values stored beside the type 'number' are always legal double-precision floating-point numbers, and values stored beside the type 'hashtable' are always pointers to properly structured hashtables in heap-allocated blocks. The value of this statement should be obvious: although it affords the programmer no compile-time protection, it makes programs easier to debug, limits the possible damage that a broken program can cause, and opens up the possibility of running untrusted programs. All this is achieved without asking the programmer to learn anything beyond what would be necessary for an untyped language.

4.1.2. Static types. One of the attractions of a dynamic type scheme is that it is very easy to check that the language is type-safe. One simply stares hard at all of the available arithmetic operations until satisfied that they are individually type-safe. The result then follows by induction on time. However, storing type information at run-time is a major expense. Performing arithmetic on type information at run-time is also expensive (although in the case of Lua this is not a problem). High-performance languages therefore invariably use some sort of *static* type scheme.

A static type scheme is one in which the run-time type of a value (or, to be pedantic, an expression, or a variable) is known perfectly at compile time, to the extent that it can be discarded at run time. An extreme example of a statically typed language is ML, which goes to great lengths to infer the type of expressions. Type inference is not typical, however, and fits badly with separate compilation. In most languages, and in ML's module system, the programmer is required to supply type annotations.

The statement which is always true is as follows: the run-time value of a variable always matches its compile-time type. Unlike in Lua, this statement is not automatically true of all programs. It is the ML compiler's responsibility to perform at compile time all the type arithmetic that Lua performs at run time, and to reject any program in which the type annotations are incorrect. If separately compiled programs are linked, the linker must perform additional type checking. Having checked all arithmetic operations *in a particular program*, the type-safety of that program follows by induction on time.

The advantages of a static type scheme include most of those of a dynamic type scheme: it limits the damage that a broken program can cause, and it supports execution of untrusted code. In addition, it is a much more effective way of debugging programs: although very few ML programs compile first time, they almost always run first time. The performance benefits are substantial: of all the world's well-known languages, OCaml (a dialect of ML) is currently the fastest safe one, beaten only by C, and is also among the most memory-efficient.

A disadvantage of a static type scheme is that some programs which at run-time behave entirely properly (according to a dynamic type scheme) will be refused by the compiler. Another is that a static type scheme is necessarily abstract, and the more abstract it gets the more difficult it is for programmers to provide the necessary type annotations. Of course, no programmer has any business writing a program unless he knows why it works; the problem is the extra language needed to express subtle reasoning. In practice, a language designer can pick a compromise between completeness and complexity.

4.1.3. Representation types. *[Omit this section? It's interesting but off-topic.]*

Java's type scheme is a mixture of many complementary ideas. For efficiency it uses static types on short time-scales, and resorts to dynamic types whenever things get too complicated. It is not an extreme example of anything, but buried within it there is a good example of representation types.

The representation of a Java object in memory is not discernible from within the language, but on modern processors it is fairly tightly constrained by efficiency considerations. It contains (among other things) a dynamic type (called its *actual class*), and a table of methods. In source code, methods are distinguished by their names, which are human-readable strings. At run-time methods are distinguished by their indices in the table, which are efficient integers. The dynamic type defines the mapping from one to the other. The methods are packed in tight to save memory, so methods of objects with different dynamic types may have different indices even if they have the same name.

Code can call the methods of an object if it has a reference to the object. The reference has a static type. The most useful static type is a perfect match for the dynamic type (a *class*), since it is then possible to find any method with only trivial arithmetic. A static type which defines a prefix of the method table (a *superclass*) is just as good, except that some methods will be unavailable. Java provides a third possibility: a static type that defines an arbitrary subset of the methods, by name (an *interface*). Calling a method through an interface is less efficient than calling it through a class, because the process of finding it is more complicated (for example, it might be retrieved from a hashtable, or from a separate table whose address must itself be calculated).

The point about representation types is that the effect of calling the method is exactly the same in all three cases.[2] Furthermore, the data structure used to represent the object in memory is exactly the same, since it depends only on the dynamic type. The only difference between the three cases is the information that is available to the caller. An implementation of Java could, if it wanted, use hashtable lookups for all method calls. It would not change the behaviour of the language. It would seriously impair its performance. The type scheme, then, is providing an optimisation mechanism that would otherwise be unavailable. It manages this while hardly requiring programmers to learn anything; they simply think in terms of the method names.

[2]At least, this is true unless there are two methods with the same name and the same number of arguments, but such that the argument types of one are subtypes of the argument types of the other, and such that the former does not appear in a superclass. Then, if the method is called through a reference whose static type is the superclass, the more specific method may be ignored. This is not a good design! Fortunately, it rarely happens.

4.1.4. Types as sets. In functional languages, and in mathematical logic, the type of an expression can be viewed as a set containing all the possible values of that expression. This picture has contributed a great deal of vocabulary, including booleans, integers, pairs, lists, collections, maps, records and so on. However it does not generalise well. It handles functions okay, but it begins to look shaky at the mention of streams, especially output streams, and falls over completely at the first attempt to describe non-functional interactive behaviour, such as that needed to grab and release a lock.

To be fair, pure functional languages like Haskell *have* found a way to cope with input and output, using monads. An interactive program is modelled as a function which acts on the whole world, and returns its new state. The type scheme is intricately designed to ensure that the output world depends on the input world via a unique dependency chain, so that it can't be shared. Intermediate values that depend on copies of the input world can exist, but ultimately only one of them can be returned.

This is ingenious but totally insane. It involves modelling the world as an element of a set, a thing that you can manipulate, while at the same time the world contains other programs that model you as a thing that they can manipulate. It escapes set-theoretic paradoxes only by using an abstract interpretation of the world. I find this philosophically very ugly.

4.2. MIN's type scheme

We have seen several reasons to have a type scheme, none of which apply to MIN. Types help to catch programming errors, but MIN is not especially aimed at human programmers. Types make programs safe, but that in itself is not a goal. Safety is merely a prerequisite for many other properties. Types can improve performance, but that is again beyond the scope of this thesis.

So why have I insisted that MIN must have a type scheme? Is it really more important than arithmetic? No, but I don't expect any reader to doubt the possibility of adding arithmetic primitives to MIN harmlessly. Similarly, I am sure you will believe that type schemes could be invented to make MIN safe, secure, efficient, or to give it any other property that type schemes have given to other languages. I have therefore gone for a very simple and conventional design: MIN's type scheme is purely static, it is logically 'simple' (in the sense of the simply-typed lambda calculus), it has no representational content, and MIN's types can't in general be interpreted as sets.

However, there is one property that is essential for a language that is designed to be optimised automatically, and which most languages do not have. It is not obvious that a type scheme can contribute to this property, and that is why I am writing a chapter about it. MIN has a type scheme because types enlarge the program equivalence relation, thereby permitting optimisations that would otherwise not be valid.

4.2.1. Program equivalence. The effect of types on program equivalence is highly counter-intuitive. After all, static types are erased before a program is run, and do not affect its operational semantics at all. However, reduction-closure is only one of the four properties that are required of program equivalence. Another is context-closure. The type checking performed by the compiler and linker substantially reduces the set of contexts in which a program might find itself, eliminating many that might otherwise provide a way of distinguishing two programs.

The point is illustrated in figure *[4A]*. Although a little contrived, this example shows the dramatic effect the types can have. All five possible equivalence relations on a three-element set are obtainable, simply by substituting different types for α. In two of the five cases (unit and stream) the equivalence relation contains an obvious optimisation that saves both time and memory, but neither transformation is valid at the other type.

Examples like this, of which there are many more in chapter 6, argue strongly that an intermediate language for an optimising compiler must have a decent type scheme, or abandon all pretence of ambition.

4.2.2. Approach.

A *typed program* consists of a *program* and an *interface*. Figure [4B] illustrates the way in which these two components are gleaned from a program in the graphical notation. It is the 'AND gate' example yet again. The program captures the rewrites and the main graph, as explained in chapter 3. The interface captures the constructor declarations, and all the type information, in a manner which I hope is obvious.

The subject-reduction property is that the program *obeys* the interface, in a sense that I will make precise in section 4.3.3. Roughly speaking, the interface defines some input transitions which the program must perform, and some output transitions which it may perform. The program and its context are treated completely symmetrically: if one may send, the other must receive.

The operation of linking two programs must be redesigned with types in mind. Not only must it construct a new program, it must also construct its interface. To ensure that the resultant program obeys the resultant interface, linking becomes a partial operation: there are perfectly legal typed programs which cannot be linked together. This is of course the whole point. The effect of types on the barbed equivalence relations follows immediately from this definition.

It is then necessary to redesign the labelled equivalence relations to keep up. Specifically, we must modify the labelled transition relation on typed programs, so as to enlarge the labelled equivalences in line with the barbed equivalences (the labelled equivalences are fairly useless unless they match the barbed equivalences). It turns out that the required modification is very simple: we must remove some of the input transitions.

4.2.3. Expressive power.

[This section is fine, but probably in the wrong place.]

Most of the proofs assume very little about the structure interfaces, allowing it to be upgraded cleanly later. However, for now I have fixed on a concrete definition of 'type' and 'interface', which I presented approximately in chapter 1, and which is used in the practical part of this thesis. My definition is based on the following simplifying approximations and restrictions:

- The ports of an interface are independent. An interface assigns each port a *type*. The transitions that may and must be possible at a port are then constrained only by its type. Furthermore, the types of any new ports created by a transition depend only on the constructor involved and the old type.
- Every port will be used either only for input or only for output. The property that distinguishes the cases is called the port's *polarity*. The polarities of new ports created by a transition depend on the constructor involved, the old type, and the old polarity. The polarity and type of a port are orthogonal; inverting all polarities is a symmetry operation.
- Silent transitions have no effect on the interface of a typed program.
- A type is an element of an abstract set, and has no internal structure. In other words, my type scheme is based on a zeroth-order logic.

Any of these decisions could be changed to obtain a more powerful type scheme. By modelling correlations between ports, we can for example specify that two copies of a value behave in the same way, that is, that the program behaves like a function (the question is moot for linear values). Bidirectional channels widen the range of programming styles that the compiler will accept. Logics with quantification support polymorphism and abstraction. I leave these possibilities for the future.

4.3. Definitions

We need to import some definitions from chapter 3. For most purposes we can treat them as black boxes, so you need not absorb all of chapter 3 just to read this chapter (of course, if this chapter is the only part of the thesis which interests you you'd be better off reading something else!).

We need the abstract sets ⟨name⟩ and ⟨constructor⟩. Recall the definition of ⟨action⟩ in terms of these:

$$\begin{array}{lll}\langle\text{action}\rangle & ::= & \langle\text{name}\rangle\,\langle\text{constructor}\rangle\,(\langle\text{name}\rangle\,,\ldots\,\langle\text{name}\rangle)\mid \\ & & \overline{\langle\text{name}\rangle}\,\langle\text{constructor}\rangle\,(\langle\text{name}\rangle\,,\ldots\,\langle\text{name}\rangle)\mid \\ & & \tau\end{array}$$

with the restriction that all ⟨name⟩s in an ⟨action⟩ must be different.

For programs, we can adopt a higher-level approach than chapter 3, and assume a set ⟨graph⟩ of programs written in the graphical notation. In the language of chapter 3, it is the quotient of ⟨program⟩ by the structural congruence. Similarly, we assume that the labelled transition relation $[\,] \xrightarrow{[\,]} [\,]$ is a ternary relation on ⟨graph⟩ × ⟨action⟩ × ⟨graph⟩. We will occasionally have to break through the abstraction layer and meddle with the concrete definitions, but not much faith is required to skip those parts.

Recall also that fn(α) is the set of ⟨name⟩s appearing in an ⟨action⟩ α, and fn(P) is the set of free names of a ⟨graph⟩ P.

4.3.1. New definitions. Assume an abstract set ⟨type⟩ of *type identifiers*. The set of *interfaces* is defined by the following grammar:

$$\begin{array}{rcl}
\langle\text{polar type}\rangle & ::= & ?\,\langle\text{type}\rangle \mid !\,\langle\text{type}\rangle \\
\langle\text{declaration}\rangle & ::= & !\,\langle\text{type}\rangle \leftarrow \langle\text{constructor}\rangle\,(\langle\text{polar type}\rangle, \ldots, \langle\text{polar type}\rangle) \\
\langle\text{signature}\rangle & ::= & \text{a set of } \langle\text{declaration}\rangle\text{s} \\
\langle\text{protocol}\rangle & ::= & \text{let } \langle\text{signature}\rangle \text{ in } \langle\text{polar type}\rangle \\
\langle\text{interface}\rangle & ::= & \text{a finite partial function from } \langle\text{name}\rangle \text{ to } \langle\text{protocol}\rangle
\end{array}$$

A ⟨polar type⟩ of the form $!t$ is an *output* type, and $?t$ is an *input* type.[3] Given a ⟨polar type⟩ T, define $\overline{t}$ be $!t$ if $T = ?t$ and $?t$ if $T = !t$. In other words, overlining a ⟨polar type⟩ toggles its polarity. Similarly define $\overline{p}$ to be let S in $\overline{T}$ if $p =$ let S in T, and define $\overline{I}$ to map each x onto $\overline{T}$ if I maps x onto T.

It is convenient to use some notational sugar. Let $x : p$ be the singleton ⟨interface⟩ which maps the ⟨name⟩ x to the ⟨protocol⟩ p. Given ⟨interface⟩s I and J with disjoint domains, let I, J be their union. For example, we can write down a three-element ⟨interface⟩ $x : p, y : q, z : r$. Let $\vec{x} : \vec{p}$ be the obvious generalisation to tuples, which maps each x_i to the corresponding p_i. It is common for all ports of an ⟨interface⟩ use the same ⟨signature⟩, so let let S in $\vec{x} : \vec{T}$ be the ⟨interface⟩ which maps each x_i to let S in T_i. I will even use this sugar for singleton ⟨interface⟩s, because I think let S in $x : T$ is more readable than x : let S in T.

Define the set fn(I) of free names of an interface I to be the domain of I.

4.3.1.1. *Variable names.* Usually, I will choose my variables according to the following convention (in addition to the convention I used for chapter 3):

- For ⟨type⟩s I will use t, u, v.
- For ⟨polar type⟩s I will use T, U, V.
- For ⟨signature⟩s I will use S.
- For ⟨protocol⟩s I will use p, q, r.
- For ⟨interface⟩s I will use I, J, K.

4.3.2. Labelled transition system on interfaces.

DEFINITION 4.3.1. (Transitions of interfaces)

Define a new labelled transition relation, a subset of ⟨interface⟩ × ⟨action⟩ × ⟨interface⟩, to be the smallest satisfying the following axioms:

$$\frac{x, \vec{y} \notin \text{fn}(I) \quad !t \leftarrow c(\vec{T}) \in S}{I, \text{let } S \text{ in } x : !t \xrightarrow{\overline{x}c(\vec{y})} I, \text{let } S \text{ in } \vec{y} : \vec{T}}\text{OUT}$$

$$\frac{x, \vec{y} \notin \text{fn}(I) \quad !t \leftarrow c(\vec{T}) \in S}{I, \text{let } S \text{ in } x : ?t \xrightarrow{xc(\vec{y})} I, \overline{\text{let } S \text{ in } \vec{y} : \vec{T}}}\text{IN} \qquad \frac{}{I \xrightarrow{\tau} I'}\text{REACT}$$

[3]This use of the symbols ! and ? is approximately copied from the programming language Pict [ref].

PROPOSITION 4.3.2. *(Transitions of interfaces)*

The labelled transition relation satisfies the following additional rules, which capture the polarity symmetry and the independence of the ports:

$$\frac{I \xrightarrow{\lambda} I'}{\overline{I} \xrightarrow{\overline{\lambda}} \overline{I'}} REV \qquad \frac{\overline{I} \xrightarrow{\lambda} \overline{I'}}{I \xrightarrow{\lambda} I'} UNREV \qquad \frac{I \xrightarrow{\alpha} I' \quad fn(\alpha) \cap fn(J) = \{\}}{I, J \xrightarrow{\alpha} I', J} PAR$$

PROOF. Easy. □

The labelled transition system on ⟨interface⟩s treats free names in the same way as that on ⟨graph⟩s: if $I \xrightarrow{\alpha} I'$ then $\mathrm{fn}(I) \vartriangle \mathrm{fn}(\alpha) \vartriangle \mathrm{fn}(I') = \{\}$, and if $\alpha = xc(\vec{y})$ or $\alpha = \overline{x}c(\vec{y})$ then $x \in \mathrm{fn}(I)$ and $\vec{y} \in \mathrm{fn}(I')$. The labelled transition system on ⟨interface⟩s is quite different from that on ⟨graph⟩s in other ways. For example, we can easily construct an ⟨interface⟩ I such that $I_1 \xleftarrow{\overline{x}b(\vec{y})} I \xrightarrow{\overline{x}c(\vec{z})} I_2$ but $b \neq c$, or $|\vec{y}| \neq |\vec{z}|$, or $I_1 \neq I_2[\vec{y}/\vec{z}]$.

4.3.3. Obeyance. The subject-reduction property that characterises MIN's type scheme is that a program cannot get past the compiler unless it obeys an interface, in the following sense:

DEFINITION 4.3.3. (Obeyance)

Say a relation $\underline{R}$ from ⟨graph⟩ to ⟨interface⟩ is an *obeyance* iff it satisfies the following:

If $P \;\underline{R}\; I$ then $\mathrm{fn}(P) = \mathrm{fn}(I)$.

If $P \;\underline{R}\; I$ and $I \xrightarrow{\lambda} I'$ then $P \xrightarrow{\lambda} P'$ and $P' \;\underline{R}\; I'$.

If $P \;\underline{R}\; I$ and $P \xrightarrow{\overline{\lambda}} P'$ then $I \xrightarrow{\overline{\lambda}} I'$ and $P' \;\underline{R}\; I'$.

If $P \;\underline{R}\; I$ and $P \xrightarrow{\tau} P'$ then $P' \;\underline{R}\; I$.

Say a pair $P :: I$ is a *well-typed program* if there exists an obeyance which relates P to I.

Note that the relation :: is both the set of well-typed programs and the largest obeyance.

4.3.4. Linking. Recall from chapter 3 that the result of linking two ⟨graph⟩s P and Q is written $P \times Q$.

DEFINITION 4.3.4. (Linking interfaces)

Define the linking operation $\times$ on ⟨interface⟩s as follows: $I, \overline{J} \times J, \overline{K} = I, \overline{K}$.

Note that linking is a partial operation; $I \times J$ is not defined for all I and J. Linking is symmetric, and the empty interface is its unit. Also, If $\mathrm{fn}(I) \cap \mathrm{fn}(J) \cap \mathrm{fn}(K) = \{\}$ and $I \times (J \times K)$ is defined then it is equal to $(I \times J) \times K$.

PROPOSITION 4.3.5. *(Linking respects obeyance)*

If $P :: I$ and $Q :: J$ and $I \times J$ is defined then $P \times Q :: I \times J$.

PROOF. Construct the relation $\underline{R} = \{P \times Q, I \times J \mid P :: I \text{ and } Q :: J\}$ and show that it is an obeyance by checking the conditions of definition 4.3.3.

Suppose $P \times Q \ \underline{R}\ I \times J$ because $P :: I$ and $Q :: J$ and that $I \times J \xrightarrow{\lambda} K'$ for some input action λ. Then either $I \xrightarrow{\lambda} I'$ and $K' = I' \times J$ or $J \xrightarrow{\lambda} J'$ and $K' = I \times J'$. Without loss of generality, assume the former. From $I \xrightarrow{\lambda} I'$ and $P :: I$ we deduce $P \xrightarrow{\lambda} P'$ and $P' :: I'$. By the PAR$'_1$ rule of chapter 3, $P \times Q \xrightarrow{\lambda} P' \times Q$. Finally, $P' \times Q \ \underline{R}\ I' \times J$ as required.

Suppose $P \times Q \ \underline{R}\ I \times J$ because $P :: I$ and $Q :: J$ and that $P \times Q \xrightarrow{\overline{\lambda}} R'$ for some output action $\overline{\lambda}$ (the case of a silent action is similar). Distinguish cases according to the rule (from chapter 3) used to derive the transition.

If $P \times Q \xrightarrow{\overline{\lambda}} R'$was derived using the PAR$'_1$ rule (the PAR$'_2$ case is similar) then $P \xrightarrow{\overline{\lambda}} P'$ and $\text{fn}(\overline{\lambda}) \cap \text{fn}(Q) = \{\}$ and $R' = P' \times Q$. This case is similar to the input case: we deduce that $I \xrightarrow{\overline{\lambda}} I'$, hence $I \times J \xrightarrow{\overline{\lambda}} I' \times J$, and $P' :: I'$, hence $P' \times Q \ \underline{R}\ I' \times J$ as required.

If $P \times Q \xrightarrow{\overline{\lambda}} R'$was derived using the COM$'_1$rule (the COM$'_2$ case is similar) then $P \xrightarrow{\overline{x}c(\vec{y})} P'$ and $Q \xrightarrow{xc(\vec{y})} Q'$ and $P' \times Q' \xrightarrow{\overline{\lambda}} R'$. From $P \xrightarrow{\overline{x}c(\vec{y})} P'$and $P :: I$ we deduce $I \xrightarrow{\overline{x}c(\vec{y})} I'$ and $P' :: I'$. Now x must be in the intersection of fn(I) and fn(J). Since $I \times J$ is defined, we cannot have $I \xrightarrow{\overline{x}c(\vec{y})} I'$ without also having $J \xrightarrow{xc(\vec{y})} J'$ for some J' such that $I \times J = I' \times J'$. Since $Q :: J$ and input transitions are deterministic, Q' must be the unique program such that $Q' :: J'$. Therefore $P' \times Q' \ \underline{R}\ I' \times J'$. By induction on the derivation of $P' \times Q' \xrightarrow{\overline{\lambda}} R'$, we deduce that $I' \times J' \xrightarrow{\overline{\lambda}} K'$ and $R' :: K'$ as required. □

We can therefore define the operation of linking two typed programs as follows:

DEFINITION 4.3.6. (Linking typed programs)
Let $P :: I \times Q :: J$ be $P \times Q :: I \times J$ if $I \times J$ is defined, and undefined otherwise.

4.3.5. Subtypes.

Say a relation $\underline{R}$ on ⟨interface⟩s is a *type simulation* iff it satisfies the following:

- If $I \ \underline{R}\ J$and $J \xrightarrow{xc(\vec{y})} J'$ then $I \xrightarrow{xc(\vec{y})} I'$and $I' \ \underline{R}\ J'$.
- If $I \ \underline{R}\ J$and $I \xrightarrow{\overline{x}c(\vec{y})} I'$then $J \xrightarrow{\overline{x}c(\vec{y})} J'$ and $I' \ \underline{R}\ J'$.

Say I is a *subinterface* of J, and write $I <: J$, iff $I \ \underline{R}\ J$ for some type simulation $\underline{R}$. In other words, $<:$ is the largest type simulation. Also, say p is a *subprotocol* of q, and write $p <: q$, iff $x : p <: x : q$ (the choice of x is immaterial).

Intuitively, (let S in) $?t$ is a subprotocol of $?u$ if it is prepared to suffer a wider variety of inputs, in much the same way that a Java subclass is one that offers more methods. Alternatively, $!u$ is a subprotocol of $!t$ if it promises to stay within a narrower variety of outputs, in much the same way that a subset is one that has fewer elements. However, neither picture completely captures the general case, in which inputs and outputs are mixed in sequence and in parallel. Arbitrarily, $?t <: !u$ for all t and u. The subinterface relation is defined so that $I <: J$ iff $\mathrm{fn}(I) = \mathrm{fn}(J)$ and $I(x) <: J(x)$ for all $x \in \mathrm{fn}(I)$.

The subinterface relation inherits a polarity-reversal symmetry. The symmetry operation is contravariant: $\overline{I} <: \overline{J}$ iff $J <: I$. Similarly, $\overline{p} <: \overline{q}$ iff $q <: p$, and in particular $!t <: !u$ iff $?u <: ?t$.

The subinterface relation (and also the subprotocol relation) is a pre-order on interfaces: it is transitive (because the composition of $<:$ with itself is a type simulation) and reflexive (because $=$ is a type simulation), but there exist different interfaces I and J such that $I <: J$ and $J <: I$. Indeed, I and J need not even be strongly bisimilar. For example, I could be let $\{t \leftarrow c(t), t \leftarrow c(u)\}$ in x :!t and J could be let $\{t \leftarrow c(t)\}$ in x :!t. It's a bit like mutual simulation. Nonetheless, if $I <: J$ and $J <: I$ then I and J are interchangeable for the purposes of type checking.

The crucial property of the subinterface relation is its interaction with the obeyance relation: If $P :: I$ and $I <: J$ then $P :: J$ (because the composition of :: and $<:$ is an obeyance).

4.4. Type checking

Recall from chapter 1 that a MIN program in the graphical notation consists of several sections. It has a main graph, and rewrite rules for executing it. These form the untyped part of the language. It also has node declarations and subtype declarations, which are the subject of this section. Finally, the main graph has an interface. The goal of this section is to show that the checks performed by the compiler are sufficient to ensure that the program obeys its interface; that is, that the type-checking algorithm is sound.

Roughly speaking, the constructor declarations and subtype declarations are used to construct a type simulation, which is then combined with the destructor declarations to construct an obeyance. Either construction can fail. In order to make a proper type simulation, the subtype declarations must be compatible with the constructor declarations. In order to make a proper obeyance, the destructor declarations must be compatible with the rewrite rules. Having checked that both constructions work, the compiler can easily check that the obeyance relates the main graph to its interface.

4.4.1. Signature.

All protocols in a MIN program written in the graphical notation have the same ⟨signature⟩. This is no handicap: given two protocols let S_1 in T_1 and let S_2 in T_2 we can choose two disjoint injections i and j on ⟨type⟩ and a form a common signature $S = i(S_1) \cup j(S_2)$, so that let S in $i(T_1)$ is equivalent to the first protocol and let S in $j(T_2)$is equivalent to the second.

The signature is written explicitly in the graphical notation, in the form of the constructor declarations. Each ⟨declaration⟩ $!t \leftarrow c(\vec{U})$ corresponds to a declaration of a constructor c whose principal port is an output of type t and whose auxiliary ports have polarities and types $\vec{U}$. For example, the program in figure *[4A]* declares the signature

$$\begin{aligned}\{!\mathrm{unit} &\leftarrow \mathrm{nil}(),\\ !\mathrm{list} &\leftarrow \mathrm{nil}(),\\ !\mathrm{list} &\leftarrow \mathrm{cons}(!\mathrm{bool}, !\mathrm{list}),\\ !\mathrm{stream} &\leftarrow \mathrm{cons}(!\mathrm{bool}, !\mathrm{stream}),\\ !\mathrm{bool} &\leftarrow \mathrm{t}(),\\ !\mathrm{bool} &\leftarrow \mathrm{f}()\}\end{aligned}$$

4.4.2. Subtype relation. The compiler constructs a type simulation from the subtype declarations in a graphical program using an 'up to' technique.

DEFINITION 4.4.1. (Type simulation precursor)

Given a relation $\underline{R}$ on ⟨protocol⟩s, let $\underline{\hat{R}}$ be the smallest relation on ⟨interface⟩s satisfying the following rules:

$$\frac{p \ \underline{R}\ q}{x : p \ \underline{\hat{R}}\ x : q} \qquad \frac{I \ \underline{\hat{R}}\ J}{\overline{J} \ \underline{\hat{R}}\ \overline{I}} \qquad \frac{I_1 \ \underline{\hat{R}}\ J_1 \quad I_2 \ \underline{\hat{R}}\ J_2}{I_1, I_2 \ \underline{\hat{R}}\ J_1, J_2}$$

$$\frac{}{I \ \underline{\hat{R}}\ I} \qquad \frac{I \ \underline{\hat{R}}\ J \quad J \ \underline{\hat{R}}\ K}{I \ \underline{\hat{R}}\ K}$$

Say $\underline{R}$ is a *type simulation precursor* iff it satisfies the following conditions (in which the choice of ⟨name⟩ x is immaterial):

- If $p \ \underline{R}\ q$ and $x : q \xrightarrow{\lambda} J'$ for input action λ then $x : p \xrightarrow{\lambda} I'$ and $I' \ \underline{\hat{R}}\ J'$.
- If $p \ \underline{R}\ q$ and $x : p \xrightarrow{\overline{\lambda}} I'$ for output action $\overline{\lambda}$ then $x : q \xrightarrow{\overline{\lambda}} J'$ and $I' \ \underline{\hat{R}}\ J'$.

PROPOSITION 4.4.2. *(Type simulation precursor)*

If $\underline{R}$ is a type simulation precursor, then $\underline{\hat{R}}$ is a type simulation.

PROOF. Induction on the derivation of $\underline{\hat{R}}$. All cases are easy. □

A program in the graphical notation explicitly includes a candidate type simulation precursor: a subtype declaration T <: U corresponds to a single element (let S in T, let S in U) of the relation (S is the program's ⟨signature⟩). Of course, the compiler must check that the declared relation really is a type simulation precursor. The checks it performs were explained informally in chapter 1. The following proposition states that they are sufficient.

PROPOSITION 4.4.3. *(Subtype checking)*

Suppose $\underline{R}$ is a relation on ⟨protocol⟩s such that whenever $p \ \underline{R}\ q$ we find either $p =$ let S in $!t$ and $q =$ let S in $!u$ or $p =$ let S in $?u$ and $q =$ let S in $?t$, and for all $!t \leftarrow c(\vec{T}) \in S$ there exists $!u \leftarrow c(\vec{U}) \in S$ such that $|\vec{T}| = |\vec{U}| = n$, say, and for all $1 \leq i \leq n$ we find either let S in $T_i \ \underline{R}$ let S in U_i or let S in $\overline{U_i} \ \underline{R}$ let S in $\overline{T_i}$. Then $\underline{R}$ is a type simulation precursor.

PROOF. Immediate from the definitions. □

4.4.3. Obeyance relation. The compiler uses another 'up to' technique to construct an obeyance.

Recall the syntax of a program from chapter 3. A graphical program is a structural congruence class, each element of which is of the form let E in p where E is an environment (a set of rewrites of the form $q \longrightarrow q'$) and p represents the main graph. The environment E is a formalisation of the rewrite rules of the program. Recall also that a wire joining x and y is written $\langle x, y\rangle$, a destructor with ports $\vec{x}$ is written $d(\vec{x})$, a halt node with free ports $\vec{x}$ is written $0(\vec{x})$, and a constructor with auxiliary ports $\vec{y}$ is attached to a name x by replacing x with $c(\vec{y})$.

DEFINITION 4.4.4. (Obeyance precursor)

Assume an environment E and a type simulation $\underline{\hat{R}}$.

Given a relation $\underline{O}$ from destructor symbols to tuples of ⟨protocol⟩s, let $\underline{\hat{O}}$ be the smallest relation from ⟨program⟩s to ⟨interface⟩s satisfying the following rules (in which I have omitted 'let E in ' everywhere):

$$\frac{d\ \ \underline{O}\ \ \vec{p}}{d(\vec{x})\ \ \underline{\hat{O}}\ \ \vec{x}:\vec{p}} \qquad \frac{P\ \ \underline{\hat{O}}\ \ I, \text{let } S \text{ in } x :?t \quad !t \leftarrow c(\vec{T}) \in S}{P[x \mapsto c(\vec{y})]\ \ \hat{O}\_\ \ I, \text{let } S \text{ in } \vec{y} : \overline{\vec{T}}} \qquad \frac{P\ \ \underline{\hat{O}}\,\underline{\hat{R}}\ \ I}{P\ \ \underline{\hat{O}}\ \ I}$$

$$\frac{\{\vec{x}\} = \text{fn}(I)}{0(\vec{x})\ \ \underline{\hat{O}}\ \ I} \qquad \frac{}{\langle x, y\rangle\ \ \underline{\hat{O}}\ \ x : p, y : \overline{p}} \qquad \frac{P\ \ \underline{\hat{O}}\ \ I \quad Q\ \ \underline{\hat{O}}\ \ J}{P \times Q\ \ \underline{\hat{O}}\ \ I \times J}$$

Say $\underline{O}$ is an *obeyance precursor* iff $\underline{\hat{O}}$ satisfies the following: for all $p \longrightarrow p' \in E$ if let E in $p\ \ \underline{\hat{O}}\ \ I$ then let E in $p'\ \ \underline{\hat{O}}\ \ I$.

Note that if $P \equiv Q$ and $Q\ \ \underline{\hat{O}}\ \ I$ then $P\ \ \underline{\hat{O}}\ \ I$ (induction on the derivation of $P \equiv Q$).

PROPOSITION 4.4.5. *(Obeyance precursor)*

If $\underline{O}$ is an obeyance precursor, then $\underline{\hat{O}}$ is an obeyance.

PROOF. The case of input transitions is easy. Suppose $P\ \ \underline{\hat{O}}\ \ I$ and $I \xrightarrow{x c(\vec{y})} I'$. Then $P \xrightarrow{x c(\vec{y})} P[x \mapsto c(\vec{y})]$ by receptivity (see chapter 3). From $I \xrightarrow{x c(\vec{y})} I'$ we know $I = I_1, \text{let } S \text{ in } x :?t$ and $!t \leftarrow c(\vec{T}) \in S$ and $I' = I_1, \text{let } S \text{ in } \vec{y} : \overline{\vec{T}}$. Finally, from $P\ \ \underline{\hat{O}}\ \ I$ we deduce $P[x \mapsto c(\vec{y})]\ \ \hat{O}\_\ \ I'$ as required.

Now for silent transitions. Suppose $P\ \ \underline{\hat{O}}\ \ I$ and $P \xrightarrow{\tau} P'$. Distinguish cases according to the derivation of $P \xrightarrow{\tau} P'$. For the PAR and COM rules, proceed along the lines of proposition 4.3.5, using induction on the derivation of the transition. Eventually, the derivation boils down to an instance of the REACT rule, and we use the fact that $\underline{O}$ is an obeyance precursor.

For output transitions, start as for silent transitions. Suppose $P\ \ \underline{\hat{O}}\ \ I$ and $P \xrightarrow{\overline{x} c(\vec{y})} P'$. As before, we can cope with the PAR and COM rules by induction on the derivation of the transition. We are left with an instance of an OUT rule. Without loss of generality, suppose it is OUT_1, so $P = \text{let } E \text{ in } \langle x, c(\vec{t})\rangle$ and $P' =$

let E in $\langle \vec{y}, \vec{t} \rangle$. The remainder of the derivation of $P \quad \underline{\hat{Q}} \quad I$ is now tightly constrained; by induction on the derivation of $\underline{\hat{Q}}$, we reduce it to a case in which $\vec{t} = \vec{z}$, a tuple of ⟨name⟩s, which is then easy. □

A program in the graphical notation explicitly includes a candidate obeyance precursor, in the form of the destructor declarations. A declaration of a destructor symbol d whose ports have protocols $\vec{p}$ corresponds to an element $(d, \vec{p})$ of the relation. The checks, described informally in chapter 1, that the compiler performs in order to ensure that the relation really is an obeyance precursor correspond exactly to its formal definition.

4.4.3.1. *Complexity.* Let me pause briefly to discuss a question provoked by the definition 4.4.4: given $\underline{O}$ and P how difficult is it to enumerate the interfaces I such that $P \quad \underline{\hat{O}} \quad I$? The answer is that it is decidable but NP-complete. The derivation of $P \ \underline{\hat{O}} \ I$ is guided by the syntax of P, which ensures that the enumeration terminates. Intuitively, it is a satisfiability problem: we're trying to choose a protocol for every wire in the main graph of P, including the internal ones, such that every node locally looks like one of its declarations.

Various factors conspire to ensure that the potentially exponential behaviour does not bite. Firstly, it is only necessary to enumerate the minimal interfaces with respect to $<:$, since the other solutions follow easily from them, and in fact don't need to be checked at all for definition 4.4.4. Secondly, the program is comprehensible to the programmer, and is probably therefore quite simple. In particular, redexes and reducts are small, and only redexes and reducts need to be checked in order to verify that a relation is an obeyance precursor. Thirdly, there will probably be several wires in a graph for which there is only one possible protocol; this permits an efficient divide-and-conquer approach. Fourthly, for the main graph, the programmer can add explicit type annotations to some internal wires if necessary.

4.4.4. Linking. According to definition 4.3.6, we can only link $P :: I$ with $Q :: J$ if I and J assign their common ⟨name⟩s exactly inverse ⟨protocol⟩s. Inverse ⟨protocol⟩s must have the same ⟨signature⟩. Since all protocols in I and all protocols in J have the same ⟨signature⟩, this means that $P :: I$ can only be linked with $Q :: J$ if the constructor declarations of the two programs are identical (or if they have no ⟨name⟩s in common). Inductively, this seems to mean that all programs running on a computer must have the same constructor declarations, and that programmers have no choice and no flexibility of data representation. Obviously this would be absurd.

The day is saved by the subtype relation. In order to link $P :: I$ with $Q :: J$ the compiler tries to find minimal interfaces I' and J' such that $I <: I'$ and $J <: J'$ and $I \times J$ is defined, and then forms $P :: I' \times Q :: J'$. Equivalently, split I into two parts I_1, I_2 so that $\mathrm{fn}(I_1) \cap \mathrm{fn}(J) = \{\}$ and $\mathrm{fn}(I_2) \subseteq \mathrm{fn}(J)$, and similarly split J into J_1, J_2. The result of linking $P :: I$ with $Q :: J$ is then $P \times Q :: I_1, J_1$, provided $I_2 <: \overline{J_2}$ (or equivalently $\overline{I_2} :> J_2$), and undefined otherwise.

The linker, as well as the compiler, must therefore understand the subtype relation. In fact, the linker must be cleverer: while the compiler operates within a program, and can use the type simulation conveniently provided by the programmer, the linker is expected to use $<:$ itself, the largest type simulation. It must construct the type simulations it needs without any help from the programmer.

This construction is not hard. The required relation is bounded above by the finite relation that relates the protocols reachable from I_2 to those reachable from J_2. Furthermore, the ports can be considered separately. Usually, the relation needed to compare two protocols will have no more elements than the smaller of the two protocols has states.

Given that conjuring type simulations from nothing is so easy, why does MIN ask the programmer to supply a type simulation precursor at all? Only to make a distinction between relationships that are intended and those that occur through coincidental name conflicts.

4.5. Program Equivalence

Having defined the set of typed programs, and the partial operation of linking them, we are in a position to define program equivalence. For the untyped calculus, we arrived at a definition of barbed coupled similarity almost by turning a metaphorical handle. Only the definition of soundness was worthy of debate. For the typed case we can use the same definition of soundness again, so there are no decisions to be made at all.

As in the untyped case, the barbed equivalence is difficult to use in practice, so we should then try to find a labelled equivalence that coincides with it. With current technology, this task is not automatic. For my simple type scheme, and using the untyped calculus as a base, it turns out to be quite easy.

4.5.1. Barbed coupled similarity. Adapting the definition of barbed coupled similarity from chapter 3 to the typed calculus is straightforward. One point of note is that programs with different interfaces are always distinguishable; it isn't even necessary to run them.

DEFINITION 4.5.1. (Barbed coupled similarity)

Let $\langle\text{constructor}\rangle'$ be larger than $\langle\text{constructor}\rangle$ by a single element 'test'. As in chapter 3, propagate this change through the definition of the calculus, so as to obtain the set $::'$ of typed programs of the enlarged calculus, analogous to the set $::$ of definition 4.3.3.

Say a relation $\underline{S}$ on $::'$, which only relates programs with the same interface, is *sound* iff for all $P ::' I \;\underline{S}\; Q ::' I$ either both P and Q have a 'test' node in their main graph, or neither do.

Say $\underline{S}$ is *reduction-closed* iff $\underline{S} \longrightarrow^* \;\subseteq\; \longrightarrow^* \underline{S}$.

Say $\underline{S}$ is *context-closed* iff for all $P ::' I \;\underline{S}\; Q ::' I$ and for all $R ::' J$ such that $I \times J$ is defined, both $P ::' I \times R ::' J \;\underline{S}\; Q ::' I \times R ::' J$ and $R ::' J \times P ::' I \;\underline{S}\; R ::' J \times Q ::' I$.

Say $\underline{S}$ is *coupled* iff $\underline{S} \;\subseteq\; \longrightarrow^* \underline{S}^{-1}$.

Say $\underline{S}$ is a *barbed coupled simulation* iff it is sound, reduction- and context-closed and coupled. Let *barbed coupled similarity*, written $\geq_C$, be the restriction to $::$ of the largest barbed coupled simulation, and let *mutual barbed coupled similarity*, written $=_C$, be its largest symmetric subrelation.

4.5.2. Labelled coupled similarity. It turns out that the right way to define the labelled transition system for typed programs is to form the intersection of the labelled transition systems on ⟨graph⟩ and ⟨interface⟩.

DEFINITION 4.5.2. (Labelled transition relation on typed programs)

Let $P :: I \xrightarrow{\alpha} P' :: I'$ iff $P \xrightarrow{\alpha} P'$ and $I \xrightarrow{\alpha} I'$.

Note that the set of actions that $P :: I$ can perform differs from those that the untyped P can perform only by the removal of some inputs; the fact that P obeys I ensures that all output and silent actions will be retained.

DEFINITION 4.5.3. (Labelled coupled similarity)

Say a relation $\underline{S}$ on :: which only relates programs with the same interface is a *labelled simulation* iff:

- If $P :: I \ \underline{S}\ Q :: I$ and $Q :: I \xrightarrow{\tau} Q' :: I'$ then $P :: I \xrightarrow{\tau}^{*} P' :: I'$ and $P' :: I' \ \underline{S}\ Q' :: I'$.
- If $P :: I \ \underline{S}\ Q :: I$ and $Q :: I \xrightarrow{\alpha} Q' :: I'$ with $\alpha \neq \tau$, then $P :: I \xrightarrow{\tau}^{*} \xrightarrow{\alpha} \xrightarrow{\tau}^{*} P' :: I'$ and $P' :: I' \ \underline{S}\ Q' :: I'$.

Say $\underline{S}$ is *coupled* iff $\underline{S} \subseteq \xrightarrow{\tau}^{*} \underline{S}^{-1}$.

Let *labelled coupled similarity*, written $\gtrsim_C$, be the largest labelled coupled simulation, and let *mutual labelled coupled similarity*, written $\approx_C$, be its largest symmetric subrelation.

There is clearly a lot of redundancy in this definition. For example, if $Q :: I \xrightarrow{\tau} Q' :: I'$ then we know $I \xrightarrow{\tau} I'$and hence $I = I'$. For another example, if $Q :: I \xrightarrow{\lambda} Q' :: I'$ and $P :: I$ then we know that a unique P' exists such that $P :: I \xrightarrow{\lambda} P' :: I'$. It would therefore be possible to concoct an equivalent but much more concise definition of a weak simulation, but I feel there is value in adhering to the familiar structure of the general case.

4.5.3. Labelled and barbed similarity coincide. This section proves the theorem that shows that the definition of the labelled transition system on typed programs is right. Specifically, typed programs are related by barbed coupled similarity iff they are related by labelled coupled similarity.

The proof closely follows the corresponding proof in chapter 3. We show that barbed coupled similarity is contained in labelled coupled similarity by showing that it is a labelled simulation, and conversely that labelled coupled similarity is contained in barbed coupled similarity by showing that it can be extended from :: to ::′ to obtain a barbed simulation. I will only present the differences.

The most expensive ingredient of the proof, namely the close relationship between reactions and silent transitions, is completely unchanged from chapter 3. However, there are two other results that need to be checked for the typed calculus. First, we must show that labelled simulations are closed under the new definition of linking. Second, we must show how to add types to the test harness used to show that barbed coupled similarity is a labelled simulation.

For the first result, we do not need the full 'simulation up to linking' result from chapter 3. The following proposition is therefore more specialised than the one it replaces.

PROPOSITION 4.5.4. *(Labelled coupled similarity is context-closed)*

Let $\underline{S}$ be the smallest relation on :: containing labelled coupled similarity $\gtrsim_C$ and satisfying the following rules:

$$\frac{P_1 :: I_1 \ \underline{S} \ Q_1 :: I_1 \quad P_2 :: I_2 \ \underline{S} \ Q_2 :: I_2}{P_1 :: I_1 \times P_2 :: I_2 \ \underline{S} \ Q_1 :: I_1 \times Q_2 :: I_2}$$

Then $\underline{S}$ is a labelled simulation.

PROOF. Suppose $P :: I \ \underline{S} \ Q :: I$ and $Q :: I \xrightarrow{\alpha} Q' :: I'$, for some $\alpha \neq \tau$ (the case $\alpha = \tau$ is similar). We need to show that $P :: I \xrightarrow{\tau}^* \xrightarrow{\alpha} \xrightarrow{\tau}^* P' :: I'$ and $P' :: I' \ \underline{S} \ Q' :: I'$. If $P :: I \ \underline{S} \ Q :: I$ because $P :: I \gtrsim_C Q :: I$ then the result is immediate. Suppose, therefore, that $P :: I \ \underline{S} \ Q :: I$ because $P :: I = P_1 :: I_1 \times P_2 :: I_2$ and $Q :: I = Q_1 :: I_1 \times Q_2 :: I_2$ and $P_1 :: I_1 \ \underline{S} \ Q_1 :: I_1$ and $P_2 :: I_2 \ \underline{S} \ Q_2 :: I_2$, and proceed by induction on the derivation of the transition.

Since $Q_1 :: I_1 \times Q_2 :: I_2 = (Q_1 \times Q_2) :: (I_1 \times I_2)$, the transition $Q_1 :: I_1 \times Q_2 :: I_2 \xrightarrow{\alpha} Q' :: I'$ was derived from $Q_1 \times Q_2 \xrightarrow{\alpha} Q'$ and $I_1 \times I_2 \xrightarrow{\alpha} I'$. Distinguish cases according to the derivation of $Q_1 \times Q_2 \xrightarrow{\alpha} Q'$. The case analysis is very similar to that in chapter 3, but I will spell it out. As usual, we may canonicalise the derivation, and so ignore the case of the IN rule.

If $Q_1 \times Q_2 \xrightarrow{\alpha} Q'$ was derived using the PAR$_1$ rule (the PAR$_2$ rule is similar) then $Q_1 \xrightarrow{\alpha} Q_1'$ and $\mathrm{fn}(\alpha) \cap \mathrm{fn}(Q_2) = \{\}$ and $Q' = Q_1' \times Q_2$. From $\mathrm{fn}(\alpha) \cap \mathrm{fn}(Q_2) = \{\}$ we know $\mathrm{fn}(\alpha) \cap \mathrm{fn}(I_2) = \{\}$, and so from $I_1 \times I_2 \xrightarrow{\alpha} I'$ we know $I_1 \xrightarrow{\alpha} I_1'$ and $I' = I_1' \times I_2$. Therefore, $Q_1 :: I_1 \xrightarrow{\alpha} Q_1' :: I_1'$. By the inductive hypothesis, $P_1 :: I_1 \xrightarrow{\tau}^* \xrightarrow{\alpha} \xrightarrow{\tau}^* P_1' :: I_1'$, and so $P_1 \xrightarrow{\tau}^* \xrightarrow{\alpha} \xrightarrow{\tau}^* P_1'$. By the PAR$_1$ rule (several times), $P_1 \times P_2 \xrightarrow{\tau}^* \xrightarrow{\alpha} \xrightarrow{\tau}^* P_1' \times P_2$.

If $Q_1 \times Q_2 \xrightarrow{\alpha} Q'$ was derived using the COM$_1$ rule (the COM$_2$ rule is similar) then $Q_1 \xrightarrow{\overline{\lambda}} Q_1'$ and $Q_2 \xrightarrow{\lambda} Q_2'$ and $Q_1' \times Q_2' \xrightarrow{\alpha} Q'$. Because $Q_1 :: I_1$ we know $I_1 \xrightarrow{\overline{\lambda}} I_1'$ and $Q_1' :: I_1'$, and therefore $Q_1 :: I_1 \xrightarrow{\overline{\lambda}} Q_1' :: I_1'$. Because $I_1 \times I_2$ is defined, we know $I_2 \xrightarrow{\lambda} I_2'$, and that $I_1 \times I_2 = I_1' \times I_2'$. Because inputs are deterministic, we know Q_2' is the unique $\langle\text{graph}\rangle$ such that $Q_2 \xrightarrow{\lambda} Q_2'$ and $Q_2' :: I_2'$, and therefore $Q_2 :: I_2 \xrightarrow{\lambda} Q_2' :: I_2'$. Applying the inductive hypothesis to $Q_1 :: I_1 \xrightarrow{\overline{\lambda}} Q_1' :: I_1'$ we obtain $P_1 :: I_1 \xrightarrow{\tau}^* \xrightarrow{\overline{\lambda}} \xrightarrow{\tau}^* P_1' :: I_1'$ and $P_1' :: I_1' \ \underline{S} \ Q_1' :: I_1'$. Similarly, $P_2 :: I_2 \xrightarrow{\tau}^* \xrightarrow{\lambda} \xrightarrow{\tau}^* P_2' :: I_2'$ and $P_2' :: I_2' \ \underline{S} \ Q_2' :: I_2'$. Therefore $P_1' :: I_1' \times P_2' :: I_2' \ \underline{S} \ Q_1' :: I_1' \times Q_2' :: I_2'$. We already know $Q_1' \times Q_2' \xrightarrow{\alpha} Q'$ and $I_1' \times I_2' = I_1 \times I_2 \xrightarrow{\alpha} I'$, so $Q_1' :: I_1' \times Q_2' :: I_2' \xrightarrow{\alpha} Q' :: I'$. Applying the inductive hypothesis a third time, $P_1' :: I_1' \times P_2' :: I_2' \xrightarrow{\tau}^* \xrightarrow{\alpha} \xrightarrow{\tau}^* P' :: I'$ and $P' :: I' \ \underline{S} \ Q' :: I'$. Finally, combine $P_1 \xrightarrow{\tau}^* \xrightarrow{\overline{\lambda}} \xrightarrow{\tau}^* P_1'$ and $P_2 \xrightarrow{\tau}^* \xrightarrow{\lambda} \xrightarrow{\tau}^* P_2'$ and $P_1' \times P_2' \xrightarrow{\tau}^* \xrightarrow{\alpha} \xrightarrow{\tau}^* P'$ to obtain $P_1 \times P_2 \xrightarrow{\tau}^* \xrightarrow{\alpha} \xrightarrow{\tau}^* P'$, and hence $P_1 :: I_1 \times P_2 :: I_2 \xrightarrow{\tau}^* \xrightarrow{\alpha} \xrightarrow{\tau}^* P' :: I'$ as required. □

For the second result, I have extracted only the part of the proof that actually concerns the test harness.

PROPOSITION 4.5.5. *(Test harness)*

Suppose a relation $\underline{S}$ *on* $::'$ *is sound, reduction-closed and context-closed, and that* $P :: I \;\underline{S}\; Q :: I$ *and* Q *does not contain a 'test' node and* $Q :: I \xrightarrow{\overline{\lambda}} Q' :: I'$ *for some output action* $\overline{\lambda}$. *Then* $P :: I \xrightarrow{\tau}^* \xrightarrow{\overline{\lambda}} \xrightarrow{\tau}^* P'' :: I'$ *and* $(P'' \times \textit{let } E \textit{ in } 1) :: I' \;\underline{S}\; (Q' \times \textit{let } E \textit{ in } 1) :: I'$.

PROOF. Suppose $\overline{\lambda} = \overline{x}c(\vec{y})$. Because Q does not contain a 'test' node, $c \neq$ test. Let $T = I(x)$ and $\vec{U} = I'(\vec{y})$, and define $J = x : \overline{T}, \vec{y} : \vec{U}$, so that $I \times J = I'$. Define a test harness $R = \text{let } E \text{ in } d(\vec{y}, x)$ where E is defined as follows:

$$\begin{array}{rcl} E & = & \{d(\vec{y}, c(\vec{z})) \longrightarrow d(\vec{y}, \text{test}(\vec{z})), \\ & & d(\vec{y}, \text{test}(\vec{z})) \longrightarrow \langle \vec{y}, \vec{z} \rangle\} \end{array}$$

and note that $R :: J$.

Applying context-closure to $P :: I \;\underline{S}\; Q :: I$, deduce $(P \times R) :: I' \;\underline{S}\; (Q \times R) :: I'$. Now $Q \times R \xrightarrow{\tau} Q_1$ where $Q_1 = Q'[\vec{y} \mapsto \vec{z}] \times \;\text{let } E \text{ in } d(\vec{y}, \text{test}(\vec{z}))$, which contains a 'test' node, and then $Q_1 \xrightarrow{\tau} Q_2 = Q'[\vec{y} \mapsto \vec{z}] \times \;\text{let } E \text{ in } \langle \vec{y}, \vec{z} \rangle = Q' \times \text{let } E \text{ in } 1$, which does not. By reduction-closure, we must therefore have $P \times R \xrightarrow{\tau}^* P_1 \xrightarrow{\tau}^* P_2$ and $P_1 :: I' \;\underline{S}\; Q_1 :: I'$ and $P_2 :: I' \;\underline{S}\; Q_2 :: I'$. By soundness P_1 contains a 'test' node but P and P_2 do not. As in chapter 3, we can examine the derivations of the transitions of $P \times R$ inductively to deduce $P \xrightarrow{\tau}^* \xrightarrow{\overline{x}c(\vec{y})} \xrightarrow{\tau}^* P' \xrightarrow{\tau}^* P''$ and $P_1 = P'[\vec{y} \mapsto \vec{z}] \times \;\text{let } E \text{ in } d(\vec{y}, \text{test}(\vec{z}))$ and $P_2 = P''[\vec{y} \mapsto \vec{z}] \times \;\text{let } E \text{ in } \langle \vec{y}, \vec{z} \rangle = P'' \times \text{let } E \text{ in } 1$. □

4.6. Further work

Chapter 6 includes examples of optimisations which require a slightly more powerful type scheme which I am not yet ready to present; these show up in its list of known sources of incompleteness. An example (not from chapter 6) of an optimisation in this class is that reversing a list twice is unnecessary, but only if the list is finite. MIN is a lazy language, and it is quite possible to write a program that returns the first few elements of a list and then loops for ever trying to calculate the next, or indeed a program that returns an infinite list. Using the more powerful type scheme, a programmer could indicate that he is only thinking of finite lists.

The modification is to introduce a *strict* version of every type. A strict output type is a subtype of the ordinary output type, and conversely a strict input type is a supertype of the ordinary input type. Syntactically, the extra subject-reduction property is that only the principal port of a constructor may have a strict output type. In other words, a port with a strict input type may not be joined to an auxiliary port. In particular, a strict input may not be joined to a non-terminating computation, but only to one that has already sent a result. It is harder to phrase this subject-reduction property in terms of the labelled transition system. Simply removing transitions is not enough. I think what is required is to replace a sequence of transitions $P \xrightarrow{\alpha} Q \xrightarrow{\beta} R$ with a single transition $P \xrightarrow{\alpha\beta} R$ if the $\xrightarrow{\beta}$ transition occurs at a strict port. This idea needs some fleshing out.

There are also examples which I don't know how to handle. For example, knowing that a program behaves functionally, in the sense that it does not keep state and does not leak data to a third party, allows the compiler to avoid calling it in cases where its result is discarded. For an even harder example, a quick-sort and a bubble-sort are equivalent only if the comparison function is total, transitive, and stateless. The problem of capturing these and similar examples is alarmingly open-ended.