Abstract

asdf

Table of contents

Chapter 1
Introduction

...

Chapter 2
Background and related work

- lots of subtyping ranting
- types / AI / dataflow connections
- type systems provide partial info (term types) and make subj. judgements
- theorycs vs compilers (preservation of programs)
- ground types as abstraction
- vars
- rc types vague desc
- n^3, PSPACE
- nom / struct
- syntax intro
- compiler
- ...

Chapter 3
Type inference as constraint solving

3.1 Constructors and Variance

Types in ⟨brick⟩ are formed via *type constructors*. A type constructor builds complex types out of simpler ones. For instance, the function type constructor ($\rightarrow$) builds function types out of an argument and a return type. So, if a and b are types, $a \rightarrow b$ is the type of functions taking an a and returning a b.

There are many type constructors other than $\rightarrow$. Others include:

- Structure types such as $\{\texttt{foo}\colon a, \texttt{bar}\colon b\}$, which is the type of structures having a field `foo` of type a and a field `bar` of type b.
- The unit type (written () in descriptions of the type system and `void` in ⟨brick⟩ source)
- User-defined (possibly generic) classes
- ###
- The special types $\top$ and $\bot$ (`any` and `none` in source code)

The type inference engine doesn't depend on the exact set of constructors. We leave the exact description of which type constructors are available and what their semantics are for ###, and describe the type inference engine generically.

Each type constructor has zero or more parameters. The parameters each have a *variance*, indicating how the constructed type changes as the parameter type changes. Covariant parameters (or ones of "positive variance") cause the constructed type to become a subtype if the parameter is replaced by a subtype, while contravariant parameters (of "negative variance") cause the constructed type to become a *super*type if the parameter is replaced by a subtype.

For instance, consider the function type $a \rightarrow b$. Imagine also that $a^{\downarrow}$ and $b^{\downarrow}$ are some subtypes of a and b respectively, and $a^{\uparrow}$ and $b^{\uparrow}$ are supertypes of a and b. Then:

$$\begin{aligned} a \rightarrow b^{\uparrow} &\geqslant a \rightarrow b \\ a \rightarrow b^{\downarrow} &\leqslant a \rightarrow b \\ a^{\uparrow} \rightarrow b &\leqslant a \rightarrow b \\ a^{\downarrow} \rightarrow b &\geqslant a \rightarrow b \end{aligned}$$

If we consider types to be predicates about the values they describe, then this is equivalent to stating that we can strengthen a statement about a function by strengthening what we say about its result, or *weakening* what we say about its parameter. ### single eq?

### fwd ref about invariant params / mutable cells

The space of type constructors is equipped with a subtype ordering, which forms a lattice. That is, for any two type constructors a and b we can form their greatest lower bound (that type constructor which is a supertype of any type constructor which is a subtype of both a and b), and their least upper bound (that type constructor which is a subtype of any type constructor which is a supertype of both a and b).

With just the type constructors $\top$, $\rightarrow$ and $\bot$, the ordering is simply $\bot \leqslant \rightarrow \leqslant \top$. The ordering for the full set of type constructors is somewhat more complex (see ###).

As well as forming a lattice, there is one extra condition attached to the space of type constructors called *convexity of arity*. It is explained fully in ###

3.1.1 $\top$ and $\bot$

The special type constructors $\top$ and $\bot$ represent the widest and narrowest types. $\top$ (written in ⟨brick⟩ source as `any`) is the common supertype of all types, the type so wide that it contains everything. Similarly $\bot$ (written in source as `none`) is the common subtype of all types, the type so narrow that it contains nothing.

$\top$ will generally arise as an underconstrained (unused) input (for instance, if a function parameter is never used then any object can be passed, hence the function requires type $\top$), or as an overconstrained output (for instance, if a function returns an integer on one line and a string on another, then nothing at all can be said about its return type, so it returns type $\top$).

Conversely, $\bot$ will generally arise as an *over*constrained input, or an *under*constrained output. If a function takes an input parameter and tries to add 5 to it, and also tries to call it, then it overconstrains the input. The input would be required to be a subtype of two incompatible types, and hence the input type is $\bot$. If a function $f(x)$ is defined as `return` $f(x)$ it will loop infinitely and not produce any output. Hence, it underconstrains its output and produces type $\bot$. (This is correct, since the result of $f(x)$ can safely be passed to any function at all with any requirements and it will not cause a type error, since it will never reach that stage).

In a type-correct and terminating program, $\top$ and $\bot$ can only appear in *dead* locations. $\top$ will be the type of dead *values*: variables that are written to but never read or parameters that are passed but never used. $\bot$ will be the type of dead *code*: functions that are never run or branches that are never taken.

3.2 Ground types

Every value in the language can be given a “ground type”. These types are those defined in the system of Amadio and Cardelli [2], extended to support the full set of type constructors.

Each ground type is a regular tree with ordered branches, where the leaf nodes of the tree are nullary type constructors (such as $\bot$,$\top$ or `int`), and the branch nodes of the are non-nullary type constructors (such as $\rightarrow$, {`field1`, `field2`}) where the children of a branch node correspond to the parameters to the type constructor.

A regular tree is essentially an infinite tree with regular structure, which allows us to encode the recursive data types necessary for object-oriented programming. For instance, an object-oriented singly-linked-list data type may have methods `insert`, `delete`, `find` (all having some function type) and a field `next`, having the same type as the list itself. Thus, we cannot represent the type of the list as a finite tree, since one subtree of the tree may refer recursively to the whole tree.

Regular trees have many equivalent definitions and representations, some of which are:

An infinite sequence of finite trees. The infinite regular tree can be approximated by a sequence of finite trees of increasing depth. Also, the regular tree can be considered to be an infinite tree with only finitely many distinct subtrees.

Directed graphs. A directed acyclic graph structure is used by many typecheckers to efficiently describe finite trees which may possibly share subtrees. If we allow cycles in the graph, the set of trees described is extended to include the regular trees

A term automaton. A finite state machine where the transitions move from a type to a part of that type (a constructor parameter) can be used to represent types in much the same way as a directed graph does. As a consequence, the set of paths in a regular tree is a regular language.

A solution to a series of unification equations. The solution to a system of equations to be solved by term unification is a regular tree in the general case. Many systems relying on unification (Prolog, ML typecheckers) restrict this to finite trees by the addtion of the “occurs check”, which disallows non-finite regular trees.

The various forms of regular trees have a few applications. The infinite and series-of-approximation views are useful in proving certain properties of regular trees, the directed graph approach is a handy implementation technique, the term automaton view is used to define subtyping between ground types and the unification representation allows us to show certain constraint graphs are satisfiable in the space of regular trees by reducing them to equality constraints.

The proofs of the various salient properties of regular trees are omitted. Detailed descriptions, including the equivalence of the above representations, can be found in [2, 8, 12, 13].

3.2.1 Equirecursive and isorecursive data types

The recursive data types encoded in regular trees are equirecursive types, rather than the more usual isorecursive types. Equirecursive types allow the direct specification of a recursive ground type: a ground type is considered exactly equivalent to its one-level unrolling. Isorecursive types on the other hand require explicit “roll” and “unroll” operations to break the recursive loop and see the recursion.

For example, consider a simple binary tree data type. It is represented by a structure with two fields[3.1]. The types of both are binary trees. In a language like Haskell with only equirecursive types, this binary tree datatype could be implemented as:

```
data BinTree = MkBinTree (BinTree, BinTree)
```

This defines a datatype `BinTree`, which is represented by a pair of `BinTree`s. The new type `BinTree` is considered distinct in the type system from its one-level unrolling `(BinTree,BinTree)`, with a roll/unroll isomorphism provided (the roll function is `MkBinTree` while the unroll function is pattern-matching, e.g. `\(MkBinTree x)->x`).

This means that data operating on `BinTree` must include a form of type annotation whenever it wants to construct items of the recursive type (a call to `MkBinTree`), and operations on recursive data-types can only be perfomed once a data declaration is made for that type.

In a language with equirecursive types, types like `BinTree` can be declared as type aliases, stating that `BinTree` is an alias for the type $\mu x.(x,\, x)$ (where μ is the standard type fixpoint operator). This implies that the type $\mu x.(x, x)$ and its one-level unrolling $(\mu x.(x,\, x),\, \mu x.(x,\, x))$ are exactly equivalent, and no roll/unroll operations are necessary to convert between them.

During object-oriented programming, heavy use is often made of recursive and mutually recursive data types. For instance, it would be very common for an object with specific responsibilities to refer to a parent object, while the parent object has a list of all child objects managed by it. Equirecursive data types mean that complex recursive object relationships can be written with no type annotations whatsoever.

The disadvantage of equirecursive datatypes is that they can ascribe a meaning to many terms which it may be preferable to consider ill-typed. For instance, self-application of a function (such as the lambda calculus term $\lambda x.x\,x$) is almost never a useful operation. The only application of such things seems to be in the definition of the Y combinator and other fixpoint operators, and it is very doubtful whether even that could be considered useful! As future work, it may be interesting to investigate extensions of the type system which allow equirecursive structure types (thus allowing lists, trees and so on to be used with little effort) while banning equirecursive function types (as these are more often than not entirely incomprehensible).

The reason for allowing equirecursive datatypes was in the end a practical one. As will be seen when the concept of closure of a constraint set is defined, it is relatively easy to determine whether a constraint set is satisfiable in the space of regular trees. However, there does not seem to be any easy way to check whether a constraint set is satisfiable using only finite trees: there is no simple analogue of the “occurs check” when dealing with inequality rather than equality constraints[13].

3.2.2 Subtyping between ground types

The subtyping relation between type constructors generalises naturally to a subtyping relation between finite ground types. If we have two ground types g and g', where

$$\begin{array}{rcl} g & = & c(l_1\colon t_1, l_2\colon t_2, \ldots) \\ g' & = & c'(l'_1\colon t'_1, l'_2\colon t'_2, \ldots) \end{array}$$

where c and c' are type constructors, l_i and l'_i are labels and t_i and t'_i are the smaller ground types of which g and g' consist, then we can determine whether $g \leqslant g'$ by checking whether $c \leqslant c'$ and whether $t \leqslant t'$ (or $t' \leqslant t$, depending on the variance of the label) for all types associated to a label that c and c' have in common.

3.1. For simplicity, I am ignoring leaf nodes here.

This notion depends on the constituent parts of g being smaller than g itself (likewise for g') in order to set up an induction proof. It is not clear how this extends to the non-finite regular trees. It seems like it should, however, given that a regular tree only has finitely many distinct subtrees and hence only finitely many pairs of (t, t') which need to be examined.

Amadio and Cardelli showed in [2] that this subtyping relation is in fact well-defined for all ground types (using an argument based on successive finite approximations to the infinite tree), and give an exponential time algorithm for deciding the subtyping relation. Kozen, Palsberg and Schwartzbach [8] defined the term automaton representation of a ground type to give an efficient $O(n^2)$ algorithm for deciding subtyping.

The subtyping relation on ground types in fact describes a lattice[13]. That is, there are narrowest common supertype and widest common subtype operations which can be defined on ground types and computed using a similar algorithm to the subtyping relation.

3.3 rc types

Ground types are insufficient for denoting the "type" of an object in ⟨brick⟩, since we do not necessarily know the exact ground type in use at each point in the program. To draw an analogy with functional programming languages such as ML and Haskell, ground types correspond roughly to monomorphic types.

The actual "type" of a term in ⟨brick⟩ may well contain free variables, which can be instantiated with any type. For instance, the identity function $\lambda x.x$ conforms to the ground types $a \to a$ for all possible ground types a. There is no single ground type which covers exactly the set of types to which $\lambda x.x$ conforms, so we need a more expressive notion of type. To this end, we allow types to contain free variables.

Some terms, as well as being usable with a range of possible types, have additional constraints. For instance, consider a "logging" version of the identity function which as well as returning its argument also prints its argument's `name` field to the screen. Hence, the function no longer works with any argument type. This version only works with argument types which contain a field called `name` of type `string`. That is, the argument's type must be a subtype of $\{\texttt{name}\colon \texttt{string}\}$.

A first attempt at typing such a function might assign it the type $\{\texttt{name}\colon \texttt{string}\} \to \{\texttt{name}\colon \texttt{string}\}$. Unfortunately, this type does not sufficiently capture the semantics of the function. In particular, if we have an object containing fields `name` and `place` and we pass it to a function of this type we would not be able to conclude that the result contained a `place` field.

What we need is to be able to specify that the argument type and the result type are the same (as in the typing for $\lambda x.x$), but that this type must be a subtype of $\{\texttt{name}\colon \texttt{string}\}$. We do this by allowing arbitrary constraints to appear in the type, giving us a notion of types known as *rc types* ("recursively constrained", the name is from [5]). The type of our "logging identity" function then becomes

$$a \to a \backslash a \leqslant \{\texttt{name}\colon \texttt{string}\}$$

The $\backslash$, read as "where", separates the type from a series of constraints on the variables defined in the type.

3.3.1 Denotation of an rc type

The free variables in an rc type means that it denotes not one but a set of ground types.

There need not be (and in the prescence of variables, often isn't) a single ground type which is a supertype of all of the set of ground types denoted by an rc type. For instance, consider the rc type $a \to b \backslash a \leqslant b$. This denotes the type of functions whose output type is a supertype of their input type, and is a possible typing for the identity function $\lambda x.x$. Its denotation includes the ground types $\text{int} \to \text{int}$, $\text{string} \to \text{string}$, $\{\texttt{f1}\colon a\} \to \{\texttt{f1}\colon a, \texttt{f2}\colon b\}$, and all of their supertypes such as $\text{int} \to \top$, $\bot \to \text{string}$, and so on. Note that there is no single ground type which captures all of these and no others: the greatest lower bound of all of these types would be $\top \to \bot$, which includes terms that don't satisfy the constraints.

Essentially, the set of ground types denoted by an rc type σ is the set of ground types that can be formed by applying an arbitrary substitution mapping the free variables of σ to ground types, as long as the constraints in σ still hold. This set is considered to be upwards-closed: the denotation of an rc type also includes all supertypes, as a term of a subtype can transparently be considered to conform to the supertype as well.

3.3.2 Subsumption

We would like to define a subtyping operation $\leqslant^{\forall}$ which captures a notion of subtyping on rc types which has the same substitutability property as the subtyping notion on ground types. That is, whenever a term of type σ is expected, and $\sigma' \leqslant^{\forall} \sigma$, a term with rc type σ' may be substituted.

This leads to a definition of $\leqslant^{\forall}$: $\sigma' \leqslant^{\forall} \sigma$ iff, for any ground type which is in ### upwards vs downwards closure....

Subsumption, unfortuneatly, does not turn out to be efficiently decidable. ### find a reference for PSPACE-hard###

3.4 Constraints and well-typedness

The process of type inference is essentially reduced to the process of building an rc type from the program source. For each operation in the program, a constraint is generated to assert that the operands support the requested operation. So, the rc type of a term includes all of the constraints necessary to ensure that the term has a valid meaning (is type-correct).

We can consider the rc type to be universally quantified: the program fragment can be considered to have *any* ground type which satisfies the constraint set.

Hence, our criterion for whether a program fragment is well-typed is to consider whether its constraint set is *satisifiable* (or equivalently, whether its denotation as defined above is non-empty). Each solution of the constraint set corresponds to a valid run of the program, and type-checking simply seeks to ensure that at least one exists. We don't solve the constraint set and pick out solutions, we merely need to prove that at least one solution exists, and then compile the program in such a way that it will work for any solution to the constraints.

3.5 Structural decomposition

###notation (subc), failure of subc (unsatisfiable constraints)

A constraint between two constructed types can be decomposed into a set of constraints between their constituent parts. For instance, suppose we have the constraint $a \to b \leqslant x \to y$. Since $\to$ is contravariant in its first parameter and covariant in its second, this constraint can be decomposed into the pair of constraints $x \leqslant a$ and $b \leqslant y$.

The conjunction of this set of constraints is equivalent to the original constraint; we have not lost any information by decomposing.

Decomposition is defined generally for all type constructors. Suppose we have two type constructors c and c' where $c \leqslant c'$ according to the subtype ordering on constructors. c takes parameters with labels draw from L, and c' takes parameters with labels from L'. I will use l_1, l_2,... to denote arbitrary elements of L and similarly for l_1',l_2', ...

We have the constraint

$$c(l_1{:}\,\tau_1, l_2{:}\,\tau_2, \ldots) \;\leqslant\; c'(l_1'{:}\,\tau_1', l_2'{:}\,\tau_2', \ldots)$$

For each $l \in L \cap L'$, let τ be the type assigned to it on the left-hand side of the constraint and τ' be the type on the right-hand side. We gain the following constraint:

$$\begin{aligned} \tau &\leqslant \tau' \,(\text{if } l \text{ is covariant}) \\ \tau' &\leqslant \tau \,(\text{if } l \text{ is contravariant}) \end{aligned}$$

For instance, the constraint $\{\texttt{field1:}\,a, \texttt{field2:}\,b\} \leqslant \{\texttt{field1:}\ c\}$ decomposes into the constraint $a \leqslant c$ since the constructor for structures with fields `field1` and `field2` is a subtype of the constructor for structures with only a `field1`. The only common label is `field1` which is covariant[3.2], and so decomposition results in the single constraint $a \leqslant c$.

`field2` is essentially ignored in the decomposition and does not appear in the decomposed set. This is the correct behaviour, as the given constraint does not in fact place any constraints upon the type b.

This ignoring of parameters that only appear on one side of the constraint has the potential to cause problems. To remedy these, a further requirement known as *convexity of arity* is placed on the structure of the constructor lattice. For all constructors c_1, c_2 and c_3 with label sets $L(c_1)$, $L(c_2)$ and $L(c_3)$ such that $c_1 \leqslant c_2 \leqslant c_3$, it must be the case that $L(c_1) \cap L(c_3) \subseteq L(c_2)$. This means that it is impossible to ignore parameters which will later be constrained: as the bound of a constraint moves up the constructor lattice, labels will never disappear and then reappear.

3.5.1 Formal definition of constructor lattice

Having defined convexity of arity, we are now able to make a formal statement of the conditions that must hold on the space of constructors. This is important, as it provides a separation between the type inference engine and the gory details of the type system and object model. The inference engine will work over any set of constructors which satisfy these properties, and the concrete types allowed can later be specified as such a set.

Firstly, a *variance* is either *positive* (or *covariant*) or *negative* (or *contravariant*), which are represented as $+$ and $-$ respectively. The set $\{+, -\}$ will be denoted $\mathbb{V}$.

A *constructor lattice* consists of:

- A set of constructors, $\mathbb{C}$ (whose elements are c_1, c_2, etc.)
- A set of labels, $\mathbb{L}$ (whose elements are l_1, l_2, etc.)
- A mapping $\mathbb{C} \rightarrow \mathcal{P}(\mathbb{L})$, called the arity of a constructor
- A mapping $\mathbb{L} \rightarrow \mathbb{V}$, called the variance of a label
- An ordering $\leqslant$ defined on $\mathbb{C}$, known as constructor subtyping

It must satisfy the following conditions:

- $\leqslant$ forms a lattice
- For all $c_1, c_2, c_3 \in \mathbb{C}$ such that $c_1 \leqslant c_2 \leqslant c_3$, $\text{arity}(c_1) \cap \text{arity}(c_3) \subseteq \text{arity}(c_2)$

Also note that since variance is a property of labels rather than of constructors, a label must have the same variance for each constructor in which it appears.

One more note on variances: the set $\mathbb{V}$ forms a monoid: the monoid of two elements, with identity $+$. Thus, we have an operation for combining variances:

$$\begin{aligned} +\cdot+ &= + \\ +\cdot- &= - \\ -\cdot+ &= - \\ -\cdot- &= + \end{aligned}$$

This expresses the notion that appearing contravariantly in a contravariant position causes a term to appear covariantly in the whole type, and will be useful for some definitions of the operations on constraint sets in the next sections, including the polarity of a variable.

Generally, a positive variance indicates an output, while a negative variance indicates an input. Terms that can be used as both outputs and inputs (e.g. mutable variables) require special treatment (see ###).

3.2. refrefref for variacne

3.6 Closure

A constraint set is *closed* if it contains all of the consequences of its constraints under transitive closure and structural decomposition. That is, a constraint set C is closed if:

- For all types a, b, c such that $(a \leqslant b) \in C$ and $(b \leqslant c) \in C$ then $(a \leqslant c) \in C$
- For all constructed types τ, τ' such that $(\tau \leqslant \tau') \in C$ then $\mathrm{subc}(\tau \leqslant \tau') \subseteq C$

###ref###

Any closed graph is satisfiable, and any satisfiable constraint set can be closed. That is, satisfiability can be checked by computing the closure of the constraint set and checking that none of the calls to subc require decomposition of an unsatisfiable constraint.

Closure computations form the core activity of the type inference engine. We maintain the constraint set in a compact form known as a constraint graph, and ensure that at all times the constraint graph is closed. Typing errors then manifest themselves by a failure to decompose a constraint, or equivalently by adding a constraint between constructed types where the left-hand

side's constructor is not in fact a subtype of the right-hand side's constructor.

Chapter 4 The type inference engine

4.1 The small terms invariant

When representing types inside the inference engine, we use Pottier's [13],[14] "small-terms invariant". That is, we avoid representing complex nested types with multiple layers of constructors and instead maintain the invariant that each type is either a variable or has only a single layer of constructors. Complex types can be reduced to this form by introducing fresh variables to link the parts of a type.

For instance, if we have the type $(a \to b) \to (c \to d)$ we would not represent it as a nested series of constructed types. Instead, we would introduce two new type variables x and y, and represent the type as $x \to y$ with the constraints that $x = a \to b$ and $y = c \to d$.

Somewhat more formally, we[4.1] define a *small constructed type* as one that is of the form $c(p, q, r, \ldots)$ where c is a type constructor and its parameters $p, q, r, \ldots$ are type variables. Any constraint set is equivalent to a constraint set where both sides of each constraint are either variables or small constructed types. The latter constraint set can be built from the former by breaking down each non-small constructed type into a small constructed type and a set of constraints.

Since our constraint sets don't support equality constraints, we might choose to represent constraints like $x = a \to b$ as a pair of constraints $\{x \leqslant a \to b, a \to b \leqslant x\}$. ### monopolarity, one constraint will do

This invariant is roughly equivalent to the standard compiler trick of representing all code in three-address form: by introducing lots of fresh temporaries, complex expression trees can be reduced to a series of statements whose expression part is of height at most one. It serves much the same purpose here as it does in an imperative language's compiler: implementation is simpler and optimisations based on finding common subexpressions are more effective (see ###).

4.2 Merging constraints

In our constraint set, we may have many, often redundant constraints on the same variable. For instance, consider a function which takes an argument (whose type is represented by the variable a) and accesses the fields `field1` and `field2` of that argument before passing it to a function which also accesses those fields. This will cause three constraints to be generated for a:

$$\begin{aligned} a &\leqslant \{\texttt{field1}\colon b\} \\ a &\leqslant \{\texttt{field2}\colon c\} \\ a &\leqslant \{\texttt{field1}\colon d, \texttt{field2}\colon e\} \end{aligned}$$

This leads to having a large number of essentially redundant constraints. Much of the information captured in the first two constraints is also given in the third, although we must be careful to keep the variables b and d (likewise c and e) separate since one may have weaker requirements than the other.

What we require is a means of combining bounds which would allow us to combine these three constraints into one.

4.1. Provisionally, see the next section

These operations for combining bounds are the lattice-theoretic greastest lower bound (written $\sqcap$) and least upper bound (written $\sqcup$). The greatest lower bound operation combines two constraints with a common LHS: if $a \leqslant b$ and $a \leqslant c$, then $a \leqslant b \sqcap c$, leading to a notion of "closest common subtype". Similarly, the least upper bound operation combines constraints with a common RHS and gives us "closest common supertype".

$\sqcap$ is used to combine constraints on inputs: the input must be of type a and also of type b, therefore it must be of type $a \sqcap b$. Similarly, $\sqcup$ combines constraints on outputs: the output may be of type a or of type b, hence it is of type $a \sqcup b$.

Since we're enforcing the small terms invariant defined above, $\sqcap$ and $\sqcup$ are quite simply defined[4.2]. We need only to extend the definition of a small constructed term to allow $\sqcap$ and $\sqcup$ to appear inside constructor parameters.

Before we give a formal description of small constructed terms, however, we must introduce polarities of terms and variables and the mono-polarity invariant.

4.3 The mono-polarity invariant

Before we can describe the optimised representation of constraint sets, we must draw a distinction between *positive* and *negative* type variables and constructed types. A type variable or constructed type "appears positively" (resp. negatively) if it is on the right-hand side (resp. left-hand side) of a $\leqslant$-constraint. Variables or types appear positively in upper bounds, and negatively in lower bounds.

This notion is extended naturally to variables appearing in the parameters of a small constructed type: if a variable appears in a covariant parameter of a type appearing positively (or in a contravariant parameter of a type appearing negatively), then it is said to appear positively. Conversely, in the other two cases a variable would be said to appear negatively. That is, in the constraint

$$a \rightarrow b \leqslant c \rightarrow d$$

the small constructed type $a \rightarrow b$ appears negatively, while $c \rightarrow d$ appears positively. The variables a and d appear positively, while the variables b and c appear negatively. More formally, consider a variable appearing in a parameter of variance v_1 to a constructed type appearing in a constraint with polarity v_2, where the variance v_1 and the polarity v_2 are both considered to be drawn from the same set $\mathbb{V}$. The variable will then be said to appear with polarity $v_1 v_2$, which is v_1 and v_2 combined via the monoid operation defined in ###.

Again, postive types or variables correspond to outputs:

##ref### It turns out to be very useful to separate those variables which are used as upper bounds (those which appear on the right-hand side of a $\leqslant$-constraint from those variables which are used as lower bounds (which appear on the left-hand side of a $\leqslant$-constraint).

We shall call the variables used as upper bounds *positive*, and represent them as a^+, b^+, ... Similarly, variables used as lower bounds will be called *negative* and represented as a^-, b^-, ... Restricting the constraint system so that each variable is only

Having established that no variable may be both positive and negative, we may now give a proper definition of variables and small constructed types. The variables are divided into two infinite disjoint sets $\mathcal{V}^+$ (positive variables, denoted a^+, b^+, ...) and $\mathcal{V}^-$ (negative variables, denoted (a^-, b^-, ...). The polarity signs are considered part of the name, there is no implied relationship between a^+ and a^-.

4.2. For a more thorough presentation of the $\sqcup$ and $\sqcap$ operators (including their generalisation to non-small terms), see [13].

Likewise, the *small constructed terms* are divided into two classes. Positive small constructed terms will be written τ_1^+, τ_2^+, ... while negative ones are τ_1^-, τ_2^-. We will occasionally abstract over the polarity of a term or variable and write a^v or τ^v to mean a term of polarity $v \in \mathbb{V}$. Similarly, $\Box^v$ will be used to abstract over $\sqcup$ and $\sqcap$, where $\Box^+ = \sqcup$ and $\Box^- = \sqcap$.

A small constructed term is defined as:

$$\tau^v = c(l_1\colon a^p \,\Box^p\, b^p \,\Box^p \ldots, l_2\colon c^q \,\Box^q\, d^q \,\Box^q \ldots, \ldots) \text{ where } p = \text{variance}(l_1) \cdot v,\, q = \text{variance}(l_2) \cdot v$$

That is, a postive small constructed term is one where the parameters to the constructor are given by a set of variables combined with $\sqcup$ in covariant positions and $\sqcap$ in contravariant positions, while a negative small constructed term uses $\sqcap$ in covariant positions and $\sqcup$ in contravariant positions.

These rules allow any set of positive small constructed terms to merged via $\sqcup$ into a single small constructed term, while any set of negative small constructed terms can be merged via $\sqcap$.

### formal def. merge

4.4 Garbage collection

A constraint set constructed from a program will include many redundant constraints. Since we are only interested in checking the satisfiability of the constraint graph, we can safely delete any constraint which can't possibly cause a failure in the closure algorithm. That is, we can safely delete any constraint which doesn't actually constrain the set of accepted programs.

The most obvious candidates for deletion are the "unreachable" constraints. Suppose we have the rc type

$$a \rightarrow b \backslash a \leqslant b, c \leqslant d \rightarrow e$$

The type variables c, d and e clearly do not constrain the type at all. We can generalise this idea a little by defining a preliminary notion of *reachability* of type variables: a type variable is reachable if it is present in the type part of the rc type (left of the $\backslash$), or if it is present in a constraint with one or more reachable variables. This notion, similar to the standard notion of garbage collection of data structures, allows us to delete the variables c, d and e and the constraints in which they appear.

For a somewhat more interesting example, consider the type

$$a \rightarrow b \backslash a \leqslant b, a \leqslant c, b \leqslant c$$

By our previous definition of reachability, the variables a, b and c are all reachable. However, the constraints $a \leqslant c$ and $b \leqslant c$ do not in fact constrain the solution space at all.

4.5 Representing the constraint set

Thanks to the garbage collection algorithm, we can vastly limit the amount of information we need to store for each variable. In particular, for any positive variable a^+ we need only store constraints of the form $\tau^+ \leqslant a^+$ and $a^- \leqslant a^+$. Since all of the constraints of the first form can be merged into a single one via $\sqcup$, all we need to store is a single positive small constructed term (written $\Omega^\tau(a^+)$) and a set of negative variables (written $\Omega^{\mathcal{V}}(a^+)$). We extend this to negative variables and end up with:

$$\begin{aligned} \Omega^\tau(a^v) &= \tau^v \\ \Omega^{\mathcal{V}}(a^v) &\subseteq \mathcal{V}^{-v} \end{aligned}$$

In the notation of [13, 14] $\Omega^\tau(a^+) = \tau^+$ would be represented as $C^\uparrow(a^+) = \tau^+$, while in [16] it would be $\tau^+ \leqslant a^+ \in K$. The advantage of this representation is that we can store half as many bounds by entirely ignoring $C^\downarrow(a^+)$, since the mono-polarity invariant ensures that it contains no information that would not be immediately removed by garbage collection.

4.5.1 Implementation detail

We now present an interesting trick for performing the garbage collection algorithm. Since the polarity of each variable is known, GC is only concerned with calculating reachability. Reachability must propagate through constructed bounds ($\Omega^\tau(a^v)$) but not through variable bounds ($\Omega^\mathcal{V}(a^v)$).

In an implementation language supporting *weak references* ### such as Haskell ###, we can implement this by using weak references to store the elements of $\Omega^\mathcal{V}(a^v)$. This causes the Haskell garbage collector to collect a variable and destroy weak references to it when it becomes unreachable except through weak references. So, this type simplification is performed automatically by the system garbage collector.

4.6 The incremental closure algorithm

When we want to add a constraint on existing variables to the constraint graph, we need to ensure that the resulting constraint graph is closed (as definined in ###). Since our garbage collection techniques and the simplification algorithms described later depend on the graph being closed, we must compute the closure of the graph incrementally.

There are four types of constraint that our constraint generation rules may produce:

$$\begin{aligned}
a^- &\leqslant a^+, \text{ for some fresh } a^- \text{ and } a^+\\
\tau^+ &\leqslant a^+, \text{ for some fresh } a^+\\
a^- &\leqslant \tau^-, \text{ for some fresh } a^-\\
a^+ &\leqslant a^-
\end{aligned}$$

The first three rules are trivial to handle: since the variables a^- and a^+ are fresh, closure is guaranteed since decomposition yields no new constraints. They are added simply by allocating space for the new variable(s) and setting Ω^τ and $\Omega^\mathcal{V}$ appropriately.

The last example is the one which requires the incremental closure algorithm. The algorithm here is the one presented and proved correct in ###citecitecite###, adapted to our constraint graph representation.

```
addConstraint($a^+$,$b^-$):
  for each $c^- \in \Omega^\mathcal{V}(a^+)$, $d^+ \in \Omega^\mathcal{V}(b^-)$:
    add $c^-$ to $\Omega^\mathcal{V}(d^+)$
    add $d^+$ to $\Omega^\mathcal{V}(c^-)$
    set $\Omega^\tau(a^+)$ := $\Omega^\tau(a^+) \sqcup \Omega^\tau(d^+)$
    set $\Omega^\tau(b^-)$ := $\Omega^\tau(b^-) \sqcap \Omega^\tau(c^-)$

  for each constraint $e^+ \leqslant f^- \in \mathrm{subc}(\Omega^\tau(a^+) \leqslant \Omega^\tau(b^-))$:
    addConstraint($e^+$,$f^-$)
```

4.7 Type simplification and optimisation

Since an rc type essentially encodes a constraint for each operation in the program source, they can grow to be very large. Due to the decomposition of constraints in the incremental closure algorithm, the size of the rc type can actually grow faster than linearly in the program size. As well as being unweildy and slow to manipulate, such large types are very difficult to understand if an error occurs or if the programmer wants to display the type of a function. Some method is needed for optimising them.

Several methods for optimising rc types have appeared in the literature ###cite###. Many of these (such as ###cite transitive collapsing###) are in fact unnecessary in this system since they are subsumed by the garbage collection done implicitly in the constraint graph. We use two other type optimisation methods, canonisation and minimisation.

Canonisation is the removal of $\sqcap$ and $\sqcup$-terms from small constucted terms in the constraint graph. As we shall see, any constraint graph is equivalent to a constraint graph (possibly with more variables) which does not include those two operations. A graph in this form will be referred to as "canonical". The algorithm is somewhat reminiscent of the algorithm to convert a non-deterministic finite state automaton into a deterministic one, by adding new states to represent sets of states in the original.

Minimisation is essentially a form of common subexpression elimination, applied to type terms to merge redundant variables. The algorithm itself is very close to Hopcroft's algorithm for minimising the number of states in a finite state automaton[1]. It expects a canonical constraint graph, and so canonisation must be run before minimisation.

Smith and Trifonov [16] defined canonisation, and Pottier [13] defined canonisation and minimisation. Our implementation is heavily based on Pottier's: the algorithms are identical, but the justification for their validity is marginally different as our constraint graphs have a simpler form.

In [13], Pottier's garbage collection does not necessarily produce a closed output constraint graph given a closed input. He then defines a weaker notion of closure known as *simple closure* and proves that it is preserved by the canonisation and garbage collection processes. In this presentation, due to application of the mono-polarity invariant globally and integration of the garbage collection algorithm with incremental closure, we know that all constraint graphs are always closed.

All of our constraint graphs are maintained in the simple form referred to in [13] as "perfect", and hence the results therein about the validity of the various optimisations (garbage collection, canonisation, minimisation) still hold for this system.

4.7.1 Canonisation

The canonisation algorithm is as follows. Introduce new variables to represent all subsets of at least two variables: $\{S^-|S \subseteq \mathcal{V}^-, |S| \geqslant 2\}$ and $\{S^+|S \subseteq \mathcal{V}^+, |S| \geqslant 2\}$. Set:

$$\begin{aligned}\Omega^\tau(S^v) &= \bigsqcap_{a^v \in S} \Omega^\tau(a^v) \\ \Omega^{\mathcal{V}}(S^v) &= \bigcup_{a^v \in S} \Omega^{\mathcal{V}}(a^v)\end{aligned}$$

Thus, S^v represents a merging of the constraints present about each $a^v \in S$. Then, each set of variables with more than two elements appearing in a constructed term may be removed and replaced with the singleton set $\{S^v\}$.

Of course, the actual implementation canonisation algorithm does not create all of the variables S^+, S^-. Instead, the new variables are created lazily as a term is found which requires them, and the actual set of new variables inserted is calculated as a least-fixed-point.

It would seem that canonisation always increases the number of terms in the constraint graph by adding more variables. This is not necessarily the case as the algorithm, while adding new variables, may cause more variables to become garbage and be removed from the constraint graph.

For instance, consider the following function:

```
def f(x) do
  x.doSomething(4)
  x.doSomething(4)
end
```

As each operation in the function introduces new constraints into the constraint graph, after inference has processed this function there will likely be two copies of the constraints which indicate that `x` must include a `doSomething` method. These constraints will be merged into a single constraint, but that constraint will include a $\sqcap$ term.

The initial constraint graph for this function might look like:

$$\begin{aligned} a^- \rightarrow b^+ &\leqslant \mathtt{f}^+ \\ a^- &\leqslant \{\mathtt{doSomething}\colon c^- \sqcap d^-\} \\ () &\leqslant b^+ \\ c^- &\leqslant e^+ \rightarrow f^- \\ d^- &\leqslant g^+ \rightarrow h^- \\ \mathtt{int} &\leqslant e^+ \\ \mathtt{int} &\leqslant g^+ \end{aligned}$$

Canonisation will introduce a new variable (call it x^-) to replace $c^- \sqcap d^-$. Its constructed bound will be the merged bounds of c^- and d^-, which will be $(e^+ \sqcup g^+) \rightarrow (f^- \sqcap h^-)$. The algorithm then introduces variables y^+ and z^- to stand for $e^+ \sqcup g^+$ and $f^- \sqcap h^-$. Thus, the final constraint graph becomes:

$$\begin{aligned} a^- \rightarrow b^+ &\leqslant \mathtt{f}^+ \\ a^- &\leqslant \{\mathtt{doSomething}\colon x^-\} \\ () &\leqslant b^+ \\ x^- &\leqslant y^+ \rightarrow z^- \\ \mathtt{int} &\leqslant y^+ \end{aligned}$$

which is actually smaller than the original graph since garbage collection removes the original c^-, d^-, e^+, g^+, f^- and h^- variables.

4.7.2 Minimisation

The minimisation algorithm in [13] grew out of other work by the same author (###) and others (###,###) based on finding variable substitutions which simplify the constraint graph while preserving its semantics.

These algorithms sought to find a substitution which mapped mutiple variables to the same variable, and so reduced the number of variables in the constraint graph. Instead of using *ad-hoc* heuristics to find this substitution, the minimisation algorithm seeks instead to find the coarsest possible equivalence relation such that any two variables which are in the same equivalence class can be merged into a single variable without changing the semantics of the graph.

An equivalence relation $\equiv$ is *compatible* with an rc type σ if, for all variables a, b in σ such that $a \equiv b$:

- a and b have the same polarity
- $C(a) \equiv C(b)$ where equivalence of small canonical constructed terms means they have the same constructor and the parameters are pairwise equivalent
- $V(a) = V(b)$

The minimisation algorithm finds the coarsest compatible equivalence relation and applies it to the constraint graph. This relation can be computed by a simple fixpoint calculation. We start off with an extremely coarse, probably incompatible relation: all of the positive variables are equivalent to each other and all of the negative variables are equivalent to each other.

Next, for each equivalence class in our tentative relation, we try to split the equivalence class by finding variables which are equivalent which cause the relation to be incompatible. We then split all such equivalence classes to resolve the compatibility.

We repeat the previous step until we can no longer split equivalence classes. Since the equivalence relations on the set of variables form a lattice under refinement, and this splitting procedure is monotonic with respect to this lattice (it always produces a finer relation), it must have a least fixed point.

Since the lattice is finite, we can be guaranteed that we eventually reach this fixpoint and so produce the coarsest compatible relation. We then apply this relation by choosing one element of each equivalence class as the representative, and replacing each variable with its equivalence class's representative. This causes all of the other variables in the graph to become garbage and be removed.

As an example, consider the following function which operates on singly-linked lists:

```
def f(x) do
  var y = x
  if true do
    y = x.next
  else do
    y = x.next.next
  end
  y = y.next
  return y
end
```

The condition on the "if" is simply `true` to avoid introducing boolean types; the types are independent of the execution path. The point of this example is that the branches of the if statement combine `x.next` and `x.next.next`, thus indicating that `x` has a recursive type. The `x.next.next` expression will lead to a complex nested type, which must be merged with the rest of the information gleaned from the program structure about `x`.

So, the ⟨brick⟩ type inferencer generates a type with many redundancies when given this function. The actual type generated, after canonisation but before minimisation has run, is:

$$\begin{aligned} a^- \to b^+ \ \setminus \ & a^- \leqslant b^+, a^- \leqslant \{\texttt{next}\colon c^-\}, c^- \leqslant b^+, c^- \leqslant \{\texttt{next}\colon d^-\}, d^- \leqslant b^+, d^- \leqslant \{\texttt{next}\colon e^-\} \\ & e^- \leqslant b^+, e^- \leqslant \{\texttt{next}\colon f^-\}, f^- \leqslant b^+, f^- \leqslant \{\texttt{next}\colon g^-\}, g^- \leqslant b^+, g^- \leqslant \{\texttt{next}\colon g^-\} \end{aligned}$$

What has happened is that the inference engine has created separate constraints for each of the unrollings of $\mu x.\{\texttt{next}\colon x\}$ used in the function. After minimisation has run, the unrollings are noticed to be compatible and the constraint graph is collapsed down to:

$$a^- \to b^+ \ \setminus \ a^- \leqslant b^+, a^- \leqslant \{\texttt{next}\colon a^-\}$$

This constraint graph is the optimal representation. For display to the user, using the techniques in ###, this type would be rendered simply as $a \to a \setminus a \leqslant \{\texttt{next}\colon a\}$.

4.8 rc type subsumption

### polarity issues in this section

### should this move?

Occasionally the question may arise of whether one rc type is a subtype of another. This is not needed for most type inferencing operations, since the closure algorithm correctly combines constraint sets ###. However, there are situations in which we need to decide this relation. In particular, a language which allows optional type annotations (which might be rc types with constraints and free variables) will need to check those type annotations against the inferred type of the program. Also, a language with some notion of an interface type which allows multiple specialised implementations of the interface (such as Haskell's typeclasses or Java's interfaces) will need to be able to check whether a given implementation of an interface in fact conforms to the type requirements of that inferface.

The closure algorithm will not suffice in this case: attempting to check whether an implementation conforms to a previously-declared interface by adding constraints to the constraint set would result in the interface being constrained to meet the implementation. We need not to constrain the interface type to fit the implementation, but simply to verify that the implementation type corresponds to the interface as it is written.

4.8.1 Entailment

A constraint graph C entails a constraint $x^- \leqslant y^+$ if, for all solutions to C the type assigned to x^- is a subtype of that assigned to y^+. This is a more subtle notion than the constraint $x^- \leqslant y^+$ simply being present in the constraint set: it is possible for the constraint not to be explicity stated but still necessarily hold in any solution.

Incremental closure is used to add a constraint to the constraint graph, whereas entailment is used to ask whether a constraint is "already there".

## example ##

Entailment is not efficiently decidable(###). However, there is an approximation algorithm (sound, but not complete) which will efficiently decide a sound, but not complete, approximation to entailment. ###.

4.8.2 Subsumption

###

As shown in ###pottier###, the garbage collection algorithm can alternatively be seen as simulating a run of the subsumption algorithm to prove that $\sigma \leqslant^\forall \sigma$, and removing all of those constraints which are not necessary for the subsumption algorithm to give a positive result.

4.9 Display

The representation of constraint graphs internal to the typechecker is designed for efficiency and ease of implementation. This representation is, however, almost entirely incomprehensible to a human programmer. In particular, the small-terms invariant results in a constraint graph which contains only small constructed terms and so requires a large number of intermediate variables which serve no purpose other than to link small terms (this is analogous to the unreadability of code where all of the expressions have been converted to three-address code).

So, we perform some simplifications before displaying a type to the user. These simplifications are not performed on the internal representation as they violate the small-terms invariant or the mono-polarity invariant in pursuit of readable type expressions. By way of examples, consider the following two functions (written in ⟨brick⟩ syntax):

```
def f1(x) do          | def f2(x) do
  x.increment(5)      |   return x
end                   | end
```

The function `f1` takes an object that has an `increment` method taking an int, and returns nothing. The function `f2` is the identity function.

The constraint graph for `f1` will look something like this:

$$\begin{aligned} a^- \to b^+ &\leqslant \texttt{f1}^+ \\ a^- &\leqslant \{\texttt{increment}\!: c^-\} \\ () &\leqslant b^+ \\ c^- &\leqslant d^+ \to e^- \\ \texttt{int} &\leqslant d^+ \end{aligned}$$

While the constraint graph for `f2` would look something like

$$\begin{aligned} a^- \to b^+ &\leqslant \texttt{f2}^+ \\ a^- &\leqslant b^+ \end{aligned}$$

While convenient for internal manipulations, these graphs are not at all readable for a programmer!

The graph of `f1` has many variables which give no information other than to split the type into small terms. For instance, c^- only exists to link the function type $d^+ \rightarrow e^-$ into the `increment` field of the argument type. Also, the variable e^- is not constrained at all (since the result of `x.increment` is never used, the programmer doesn't care about the variable used to represent its type).

The graph of `f2` has a slightly different problem: to comply with the mono-polarity invariant, we must use different type variables to describe the argument and the result of the function. The programmer has no such qualms, and would prefer to see the type as the more natural $a \rightarrow a$.

Both of these type graphs can be made readable by applying a simple substitution: replacing type variables with their unique bounds. ### def in terms of graph, cite, safety, varvar

This applies the following substitutions to $\texttt{f1}^+$:

$$\begin{array}{rcl} \texttt{f1}^+ & \mapsto & a^- \rightarrow b^+ \\ a^- & \mapsto & \{\texttt{increment}\colon c^-\} \\ b^+ & \mapsto & () \\ c^- & \mapsto & d^+ \rightarrow e^- \\ d^+ & \mapsto & \texttt{int} \\ e^- & \mapsto & \top \end{array}$$

Thus, the type of `f1` is displayed to the user as

$$\{\texttt{increment}\colon \texttt{int} \rightarrow \top\} \rightarrow ()$$

which is much more readable. Similarly, the following substitution is applied to $\texttt{f2}^+$:

$$\begin{array}{rcl} \texttt{f2}^+ & \mapsto & a^- \rightarrow b^+ \\ b^+ & \mapsto & a^- \end{array}$$

The choice of $b^+ \mapsto a^-$ was arbitrary, the algorithm could equally have been the other way around. So, the type of `f2` is displayed to the user (without polarity indicators) as:

$$a \rightarrow a$$

Chapter 5
... (objects, nom/struct, constraintgen)

5.1 Generalised and ungeneralised bindings

A binding of a name may be *generalised*. This allows it to be used with multiple different incompatible types at different points in the program. ### xref ### For instance, consider this function:

```
def id(x) do
  return x
end
```

This function has type $a \to a$, for all values of a. It may be used with different instantiations of this type scheme at different points in the program: in one instance it may be passed a string and return a string, and in another it may be passed an integer and return an integer.

Generalisation essentially means that the type of the term is inferred based on its definition but not on its uses. Each use must be compatible with the definition, but the uses need not be compatible with each other.

An *ungeneralised* binding, on the other hand, infers its type based on both its definition and its uses. In the example above, if `id` were called with both string and integer arguments at different points in the program, it would be inferred the type $\top \to \top$. That is, its argument would be of any type, and it would return an argument about which nothing could be proven.

Typing ungeneralised bindings is simpler as all of the information about the binding can be merged into a single set of constraints. Generalised bindings offer more flexibility since they allow terms to be used in different ways at different points in the program. Unfortuneatly, we can't generalise everything: type inference with first-class generalised bindings is undecidable[5.1]. The variables bound as function arguments must therefore be ungeneralised, and as we'll see in the next section, due to imperative constructs some other classes of binding must be ungeneralised.

5.1.1 The value restriction

In languages like Haskell, every `let`-bound and all toplevel bindings are generalised[5.2]. This poses a well-known problem[17, ?] in the presence of mutable references and side-effects. Consider this example:

```
def obj = {list = []}
```

This creates an object containing a single mutable field which is an empty list. If we generalise the type, we infer that `obj.list` is a list of element type a. That is, it may be used with any possible element type.

This causes a problem when we refer to `obj`. If we store an integer into `obj.list`, the typechecker will instantiate a as `int`, and the program will pass the typechecker. If we read a string from `obj.list`, the typechecker will instantiate a as `string`, and the program will pass the typechecker. But the program will crash since what's written as an `int` can't be read as a `string`!

5.1. Although, by requiring type annotations in situations requiring first-class generalised bindings, the typing problem can be made tractable, see [?, ?] for examples.

5.2. Generalised bindings are known as `let`-bindings in functional languages.

The problem is that having mutable references allows communication between different uses of a binding. Thus, the uses of objects containing mutable fields must be compatible, and hence the binding can't be generalised.

There are a number of standard techniques used to mitigate this problem. Tofte's system[15], used in many ML implementations, separates the type variables into two categories: the *imperative* and the *applicative* type variables. A binding will not be generalised if it contains imperative type variables. There are various increasingly complex extensions of this system, such as Leroy's system[10], which all aim to generalise as many bindings as possible. They have the property that any purely functional term can be generalised, as is the case in languages without direct imperative features.

The *value restriction*[17, 18] is a much simpler alternative. Using it, only values (that is, literal constants, functions or immutable data structures consisting only of other values) may be generalised in a binding. This results in a certain loss of generality: some terms which could previously be generalised cannot with this restriction in place. However, simple changes to such terms (making them functions, essentially) make them generalisable, so it seems to be worthwhile for the reduction in complexity of the type system compared to other solutions.

Finally, the value restriction is much more natural in ⟨brick⟩, a language where imperative constructs are pervasive. Since almost all terms include some imperative side-effects, separating imperative and applicative type variables would have little benefit as purely applicative typings would be difficult to achieve.

In fact, ⟨brick⟩ adopts an even more restrictive version of the value restriction, on the basis that it should be easier to understand: only function bindings (those of the form "`def f(x)`") and classes are generalised.

5.2 Optional type annotations

Due to the type inference system, ⟨brick⟩ programs need not provide explicit type annotations. Generally, this is an improvement over fully-annotated code as the signal-to-noise ratio increases and less of the code consists of repetitive type declarations.

However, often it is valuable to be able to provide some annotations, as a form of machine-checked documentation. This is common practice in the Haskell programming language (which also supports global type inference), where annotations are commonly placed on top-level functions as documentation of the interface provided.

Annotations may also be used to voluntarily restrict a piece of code with a very general type to a specific subtype. By the behavioural subtyping rule, a term with a given type can also be considered to have any supertype of that type. A programmer, after writing a program fragment with a particular type t, may want to explicitly limit the uses of that fragment as though it in fact had type t', where $t \leqslant t'$. This could be done to make it more clear what the programmer's intentions were and to indicate to a user what the function's purpose is.

As an example, consider a program which uses the type $\{\texttt{x}\colon \texttt{float}, \texttt{y}\colon \texttt{float}\}$ to represent points on a 2-D plane. The programmer writes a function `get_x` to extract the `x` co-ordinate of a point[5.3]:

```
def get_x(point) do
  return point.x
end
```

This will be given the type $\{\texttt{x}\colon a\} \to a$ by inference. However, the programmer wishes to indicate that this function is to be used only with points, and not with merely any structure which declares an x coordinate. So, the function can be rewritten to include specific annotations:

```
def get_x(point:{x:float,y:float}):float do
  return point.x
end
```

In this case, the function has type $\{\texttt{x}\colon \texttt{float}, \texttt{y}\colon \texttt{float}\} \to \texttt{float}$, which better indicates the programmer's intentions. Since $\{\texttt{x}\colon \texttt{float}, \texttt{y}\colon \texttt{float}\} \to \texttt{float}$ is a supertype of $\{\texttt{x}\colon a\} \to a$, the annotation is valid.

5.3. Ignoring for a moment that `get_x` is actually *longer* than the expression it abstracts!

5.2.1 Checking type annotations

For reasons outlined in the introduction###, it was a design goal to support both structural and nominative typing.

⟨brick⟩ is not the first language to seek to combine nominative and structural typing for objects. Whiteoak[6] is an extension of the Java programming language with, amongst other features, structural subtyping. Their implementation of structural subtyping is somewhat complex, as it is necessary to shoehorn the new feature into the existing nominatively-typed Java virtual machine. Since Java requires full type annotations, this is done by detecting all points in the program where a nominative type is converted to a structural type and generating code for wrapper objects at rutime.

A similar effort was undertaken to add structural subtyping to the language Scala[4] which also runs on the Java virtual machine. This included a more sophisticated implementation, combining techniques based on runtime code generation and Java's reflection mechanism.

The Unity language[11] combines nominative and structural subtyping in a way somewhat closer to the technique used in ⟨brick⟩, as well as providing other features such as external dispatch (aka multi-methods) and a full proof of soundness. It does not attempt to tackle the problem of integrating type inference with this system.

### footnote that multiple subtyping is vital, regardless of whether multiple inheritance is a good thing ###

Here is a concrete example: Suppose we have class `Cowboy` and class `Shape`. Both of these have a method `draw`, with very different meanings. Class `Square` is a subtype of `Shape`, implementing the `draw` defined by shapes.

Now suppose we define three functions:

```
def drawany(x) do          | def render(x: Shape) do    | def brandish(x: Cowboy) do
  x.draw()                 |   x.draw()                 |   x.draw()
end                        | end                        | end
```

We also have three variables `a`, `b`, and `c`, of types `Cowboy`, `Shape`, and `Square` respectively. We'd like them to be compatible with the functions defined above according to this matrix:

	`a: Cowboy`	`b: Shape`	`c: Square`
`drawany`	✓	✓	✓
`render`		✓	✓
`brandish`	✓		

That is, the functions that required their arguments to be of specific class types only accept those types, while the function that required merely that its argument have a method called "draw" accepts the structural type "anything that has a draw field".

To fit this into the type inference system we need to define the subtyping relation between object types. It is clear from the above that we should have `Shape` and `Cowboy` be subtypes of `{draw}` (i.e. the structural type "containing a draw field"), but unrelated to each other.

5.2.2 A potential problem

A class may extend any number of other classes. Whether multiple inheritance (the ability to inherit code and data definitions from multiple extended classes) is a good thing or not, it is clear from its presence in almost every that multiple *subtyping* (the ability to implement interfaces from multiple exteded classes, such as Java or C# interfaces) is vital to a nominatively-typed object-oriented language. It is this feature which allows, say, a `List` class to be iterated over (via a `Iterable` interface), checked for equality with other lists (via a `Comparable` interface) and so on using common generic interfaces.

So, consider two subclasses of `Shape`: `Rectangle` and `RegularPolygon`. `Rectangle` defines width and height fields, and `RegularPolygon` defines a side-length field. A subclass `Square` is created, extending both of these. This poses no problem: `Square` must simply implement all of the interfaces its superclasses demand. That is, it must provide `draw`, `width`, `height`, and `side-length`.

What if a programmer tried to write a subclass of both `Rectangle` and `Cowboy`? In particular, what operation would such a class's `draw` method perform? One of the central features of a nominative type system is that two features of a type are not considered equivalent merely by having the same name.

The problem is not merely having two different superclasses that define the same name. There was no problem with `Square` having a `draw` method, even though it was part of both `Rectangle` and `RegularPolygon`, since both methods referred to the same *meaning*, that of `Shape.draw`. Our hypothetical rectangular cowboy has no such luxury: its `draw` method must implement both `Shape.draw` and `Cowboy.draw`, which is nonsensical.

We cannot allow such objects: we cannot demand that there be a single `draw` method, for that would violate the principle that nominative declarations are not equivalent unless declared so, and we cannot allow multiple `draw` methods since we cannot in general disambiguate (what if such an object was passed to `drawany` above?).

So, the problem is solved simply by disallowing such objects. This requires a certain amount of subtlety: we must find a way of detecting when a constraint graph requires such an impossible object and consider it in error.

Having given an intuitive explanation for how we would like nominative and structural types to interoperate, it remains to fit it into the formal model of constructor lattices defined in section ###. If this can be done, then we will know that the type inference algorithms will support inference and annotation-checking of code using these types.

So, we would like to add *object types* to our constructor lattice. These types must support an arbitrary set of fields, and must support (user-defined) classes. As was explained above, a class is a subtype of the purely structural type with the same set of fields, as well as being a subtype of those classes it explicitly extends. An important (almost defining) property of nominative typing is that two classes do not enter into a subtype relation merely by having the same set of fields. However, those two classes must both be subtypes of the structural type defining their common fields.

5.2.3 Classes (partial description)

We may consider a class, therefore, as consisting of a set of classes that it directly extends and a set of members that it defines (as specified by the programmer in the class definition). Classes will be written as $\boldsymbol{C}_1$, $\boldsymbol{C}_2$, ... with $\boldsymbol{C}_1$ **extends** $\boldsymbol{C}_2$ denoting the "directly extends" relation and **defined**($\boldsymbol{C}_1$) denoting the set of members defined in the class.

We will temporarily ignore that part of the class which places restrictions on the types of its members, which will be explained in a later section. For now, we seek only to define a constructor lattice capable of representing the top-level types.

The directed graph formed by the **extends** relation is acyclic, and so its reflexive transitive closure forms a partial order, which we will refer to as *subclassing*[5.4]. $\mathbf{superclasses}(\boldsymbol{C})$ will denote the set of classes of which $\boldsymbol{C}$ is a subclass.

As well as defining some members, a class inherits the members defined in each of its superclasses. Some of these members (particularly methods) may be overridden and given a different definition in the superclass, giving rise to "polymorphic dispatch": the target of a function call depends on the runtime class of a object ### reword ###

We will consider $\mathbf{defined}(\boldsymbol{C})$ to include only those members defined for the first time in $\boldsymbol{C}$, rather than those inherited or overriden. Thus, we can define $\mathbf{members}(\boldsymbol{C})$, the complete set of members in $\boldsymbol{C}$, as $\bigcup\,\{\mathbf{defined}(\boldsymbol{C'})|\boldsymbol{C'} \in \mathbf{superclasses}(\boldsymbol{C})\}$.

The sets $\{\mathbf{defined}(\boldsymbol{C'})|\boldsymbol{C'} \in \mathbf{superclasses}(\boldsymbol{C})\}$ must be disjoint, and so for any member in $\mathbf{members}(\boldsymbol{C})$ we may find the unique superclass $\boldsymbol{C'}$ which defines it. Two classes which define disjoint sets of members are said to be *compatible*, and so the above condition may be restated as "no class may have two incompatible superclasses". In the example above, `Shape`, `Regular-Polygon` and `Rectangle` are all pairwise-compatible (and so `Square` is a legal class), but `Shape` is not compatible with `Cowboy` (thus banning rectangular cowboys).

Since the set of classes extended and the set of members defined by a new class are written directly in the class declaration, these restrictions (that **extends** is acyclic and that superclasses are all pairwise compatible) are simple syntactic criteria and can be verified without invoking the typechecker.

This brings us part-way towards solving the problem mentioned above: it now becomes impossible to define a class which is a subclass of both `Cowboy` and `Shape`, since the `draw` method would not have a unique definition. The problem is not completely solved, however, as it will require some more sophistication to recognise that a function which tries to use objects of both these types (say, a function that passes its argument to both `brandish` and `render`) is ill-typed.

5.2.4 The object constructor lattice

An *object type constructor* $\boldsymbol{O}$ consists of a set of classes $\mathbf{classes}(\boldsymbol{O})$ and a set of fields $\mathbf{fields}(\boldsymbol{O})$ such that:

- $\mathbf{classes}(\boldsymbol{O})$ is upwards-closed.
  That is, for all $\boldsymbol{C} \in \mathbf{classes}(\boldsymbol{O})$, $\mathbf{superclasses}(\boldsymbol{C}) \subseteq \mathbf{classes}(\boldsymbol{O})$.
- The elements of $\mathbf{classes}(\boldsymbol{O})$ are compatible.
  That is, for all $\boldsymbol{C_1}, \boldsymbol{C_2} \in \mathbf{classes}(\boldsymbol{O})$, $\boldsymbol{C_1}$ and $\boldsymbol{C_2}$ are compatible.
- $\mathbf{fields}(\boldsymbol{O})$ contains at least the fields defined by the classes in $\mathbf{classes}(\boldsymbol{O})$.
  That is, for all $\boldsymbol{C} \in \mathbf{classes}(\boldsymbol{O})$, $\mathbf{members}(\boldsymbol{C}) \subseteq \mathbf{fields}(\boldsymbol{O})$.

The space of object type constructors $\mathbb{O}$ consists of all such object type constructors, as well as a bottom element $\bot_{\mathbb{O}}$. For the type inference engine to successfully check programs, it must be proven that $\mathbb{O}$ forms a type constructor lattice (as defined in ###).

- The set of constructors is $\mathbb{O}$.
- The set of labels is the set of all possible field names.
- $\mathrm{arity}(\boldsymbol{O}) = \mathbf{fields}(\boldsymbol{O}); \mathrm{arity}(\bot_{\mathbb{O}}) = \{\text{all possible field names}\}$
  The arity is simply the set of fields, each field becomes a label.
- $\mathrm{variance}(f) = +$
  All of the labels (fields) have positive variance.###
- $\bot_{\mathbb{O}} \leqslant \boldsymbol{O}; \boldsymbol{O_1} \leqslant \boldsymbol{O_2} \Leftrightarrow \mathbf{classes}(\boldsymbol{O_1}) \supseteq \mathbf{classes}(\boldsymbol{O_2}) \wedge \mathbf{fields}(\boldsymbol{O_1}) \supseteq \mathbf{fields}(\boldsymbol{O_2})$
  $\bot_{\mathbb{O}}$ is a subtype of every type constructor.

5.4. Not that, under this definition, each class is a subclass of itself.

A type constructor $\boldsymbol{O_1}$ is a subtype of $\boldsymbol{O_2}$ if it has a larger set of classes and fields.

We need to show that the ordering on object type constructors forms a lattice and follows the convexity of arity condition. The latter is easy: the constructor ordering is anti-monotonic in arity (that is, $\boldsymbol{O_1} \leqslant \boldsymbol{O_2}$ only if $\text{arity}(\boldsymbol{O_1}) \supseteq \text{arity}(\boldsymbol{O_2})$), and so the convexity condition trivially holds.

It remains to show that this structure forms a lattice. Rather than prove this directly (which would be somewhat messy), we prove it by ###.

First, a few well-known lattice constructions:

- If S is a set, then S forms a lattice under $\subseteq$:
  $a \sqcap' b \,\text{is}\, a \cap b;\, a \sqcup' b \,\text{is}\, a \cup b;\, \bot' \,\text{is}\, \varnothing;\, \top' \,\text{is}\, S$
- If S is partially ordered, then the upwards-closed subsets of S form a lattice under $\subseteq$:
  $a \sqcap' b \,\text{is}\, a \cap b;\, a \sqcup' b \,\text{is}\, a \cup b;\, \bot' \,\text{is}\, \varnothing;\, \top' \,\text{is}\, \mathcal{P}(S)$
- If L_1, L_2 are lattices, then $L_1 \times L_2$ forms a lattice where $(x, y) \leqslant (x', y') \Leftrightarrow x \leqslant x'$ and $y \leqslant y'$:
  $(a, x) \sqcap' (b, y) \,\text{is}\, (a \sqcap b, x \sqcap y);\, (a, x) \sqcup' (b, y) \,\text{is}\, (a \sqcup b, x \sqcup y);\, \bot' \,\text{is}\, (\bot, \bot);\, \top' \,\text{is}\, (\top, \top)$
- If L is a lattice, the dual of L (the same set with the ordering reversed) forms a lattice:
  $a \sqcap' b \,\text{is}\, a \sqcup b;\, a \sqcup' b \,\text{is}\, a \sqcap b;\, \bot' \,\text{is}\, \top;\, \top' \,\text{is}\, \bot$

Lemma 5.1. *If L is a lattice, $s \in L$ and $\text{erase}_s(L) = \{x | x \in L, x \nleqslant s\} \cup \{\bot\}$, then $\text{erase}_s(L)$ is a lattice.*

Proof. Let $L' = \text{erase}_s(L)$. It must be shown that L' is partially ordered, and has unique l.u.b and g.l.b.

The elements of L' are a subset of those of L (with all the non-bottom elements below s removed). The partial ordering on L' will be L's ordering restricted to the elements of L', so L' is trivially a poset.

Let x, y be arbitrary elements of L'. If either x or y is $\bot$, then $x \sqcap y$ and $x \sqcup y$ are trivially well-defined. Hence we can assume x and y are not $\bot$, and by definition of erase_s that $x \nleqslant s$ and $y \nleqslant s$.

Let $a \in L = x \sqcup_L y$. Since $x \nleqslant s$ and $y \nleqslant s$, $a \nleqslant s$ and so $a \in L'$. Thus, $\sqcup$ is well-defined on L'.

Let $b \in L = x \sqcap_L y$. □

$\text{erase}_s(L)$ can be thought of as a new lattice with all elements below s removed. We can now restate the object type structure in terms of lattices: Define the full class-lattice as the upwards-closed subsets of the set of classes under the subclassing relation[5.5]. Define the class-lattice as the full class-lattice with each pair of incompatible classes removed via erase_s

######

The requirement that our types form a lattice led to the inclusion of a few extra types (to "fill in the blanks" by giving every pair of types a glb and lub), some of which turn out to be independantly useful. In particular, we gain intersection types: for any two classes or interfaces C_1 and C_2, it is possible to define a function which takes arguments of type $C_1 \sqcap C_2$###, demanding that the parameter implement both of these interfaces. This is a useful property that cannot be expressed in many statically-typed languages, including Java[5.6].

5.5. where the subclassing relation is the reflexive transitive closure of the relation given each classes' listed parent classes, that is it represents "indirect subclassing"

5.6. It is possible to define a new interface type which extends both C_1 and C_2, but then both classes must be modified to explicitly implement it, something which may not be possible if the interfaces are from different packages.

5.2.4.1 Mutability

Should mutable type constructors (e.g. array, structure) be co- or contra-variant? Neither seems to work in the presence of mutability. For example, Java chose arrays to be covariant, allowing the following code:

```
void f1(Mammal[] animals){
  mammals[0].eat();
}
...
Dog[] dogs;
f(dogs);
```

We can safely pass an array of `Dogs` to a function expecting an array of `Mammals`. Now consider this example:

```
void f2(Mammal[] mammals){
  mammals[0] = new Cat();
}
...
Dog[] dogs;
f(dogs);
```

This is a type error! (It will in fact be accepted by the Java compiler but will lead to an `ArrayStoreException` at runtime). An array of `Dogs` cannot be considered to be an array of `Mammals` when it is being stored into: you cannot store just any mammal into an array of dogs.

The solution to this adopted by many languages is to introduce *invariant* type constructors. That is, there will be type constructors `C[t]` where `C[t1]` and `C[t2]` are never related by subtyping unless `t1` and `t2` are exactly the same type. This is the solution adopted by Java's generics.

This is a little problematic for us: Firstly, `f1` was an entirely sensible piece of code and it would be better to allow it, and secondly adding invariant type construtors would violate the assumption in the type system that all type constructors are co- or contra-variant.

The problem arises from the type parameter being used as covariant during "read" operations (as in `f1`), and contravariant during "write" operations (as in `f2`). Following ###ref###, we solve the problem by introducing a pair of type parameters: one covariant and one contravariant#######

5.2.4.2 "Polymorphism"

"Polymorphism" in general means using the same code at different types. There are two distinct notions of polymorphism in common use: subtype polymorphism, common in OO languages and parametric polymorphism, common in functional languages.

Subtype polymorphism allows the use of an object with a subtype whenever a supertype was expected. For instance, a function that calculates the area of any shape would require that its argument be of type `Shape`. However, it can be passed a `Square`, as all `Squares` are `Shapes`. Subtype polymorphism essentially amounts to transparently strengthening preconditions or weakening postconditions.

Parameteric polymorphism allows the use of an object whose type is only partially specified. For instance, a function returning the first element of a list will work for all possible types of list. It can be considered a function from lists of integers to integers, or equally a function from lists of strings to strings.

Both are useful and serve broadly different goals, as evidenced by the (mostly successful) addition of parametric polymorphism to imperative languages (generics in Java and C# being good examples) and the (variously successful attempts at) adding subtype polymorphism to functional languages.

In ⟨brick⟩, parametric polymorphism manifests itself as a type with variables. The list example above would have type `List`$[a] \rightarrow a$ for all a. Subtype polymorphism is implicit in the typing rules, and the shape area example above would have type `Shape` $\rightarrow$ `float` (which can transparently be used as if it had type `Square` $\rightarrow$ `float`).

Chapter 6
Implementation tools

6.1 Haskell

The pure, lazy, functional language Haskell ###cite### was used for the implementation. There are two somewhat unusual features of Haskell which were used[6.1] heavily to describe the generic compiler infrastructure: laziness and monads.

6.1.1 Laziness

Haskell evaluates lazily. That is, when evaluating an application of a function to a value, Haskell will go straight into the evaluation of the function and only evaluate the value as and when it's needed. This means that a number of constructs which would loop infinitely in other languages complete in a finite time in Haskell. Also, constructs which would cause an error such as division by zero or the built-in `error` function only propagate the error if the offending value is actually examined during the computation. For example:

```
f x y = x * 2
```

This defines a function `f` which takes two arguments and returns the first multiplied by two, ignoring the second. Haskell's laziness means that a term like `f 42 (error ‘‘broken!’’)` will in fact output 84, since the error term is never evaluated.

We can do some more interesting things with laziness. In particular, it allows us to define "infinite" data structures:

```
biglist = [1,2,3] ++ biglist
```

Here `biglist` represents the list `[1,2,3,1,2,3,1,2,3,1,2,3,...]`. This list is evaluated as needed, and so takes only a finite amount of memory. Similarly, we can "use" the result of a computation before it has been fully evaluated:

```
let result = f 100
    f x = [1,2] ++ [a + x | a <- result]
in result
```

That is, the result is defined to be 1, then 2, followed by each element of the result with 42 added. When evaluated, the list comes out as `[1,2,101,102,201,202,...]`. Of course, we must ensure that we don't try and use a value in the computation of that value, but we may use the earlier parts to compute the later parts. For instance, if this example did not include the `[1,2]` at the start, there would be no way to compute the first element of the list and so the program can hang[6.2]. This tactic is used in the implementation of recursive functions, loops and similar recursive structures.

6.1.2 Monads

Monads are a central mechanism for expressing sequential code in Haskell. There are many, many, many introductions to the concept (###) ranging from category-theoretic interpretations to their use in describing I/O effects. What follows is a broad overview of how they can be used to describe sequential computation in a generic way, without reference to the technicalities of Haskell's type system. For a deeper discussion, see any of the above references.

6.1. Abused

6.2. Such errors are known as "strictness bugs" and are some of the most truly evil problems to debug as attempting to observe the value will change the order of evaluation.

Haskell does not have any notion of "side-effect" as present in most other languages. When a Haskell function is invoked, the only thing that can happen is the function producing a value. The function cannot modify a global, or perform I/O, or change a local variable, or perform side-effects of any form. This has advantages (you can be utterly sure that the function doesn't silently change some important piece of state) but it makes many constructs which are trivial in an imperative language difficult to express in Haskell.

Haskell supports pieces of imperative-looking code like:

```
do {x <- doSomething 42;
    somethingElse x;
    return (y + 7)}
```

This does not execute as would be expected by someone familiar with imperative languages. In an imperative language, the semicolon acts as a sequencing operator separating evaluations, so that whatever side-effects the first computation has can affect the second.

In Haskell there are no side-effects and order of evaluation is irrelevant. So, the semicolon must do something different. Haskell's semicolon is *programmable*: exactly what it does depends on which monad the statement is being evaluated in.

Semicolon is a sequential composition operator: its action is to combine two statements into a larger statement. The second statement may depend on the results of the first, as in the call to `somethingElse` above and can so be considered a function producing a statement from a value. Thus, a semicolon takes a statement on the left, and a function from values to statements on the right, and combines them into a larger statement whose effect is to perform both statements, passing the result of the first into the second.

This allows a number of otherwise difficult-to-express operations to be written simply. For instance, some monads in the Haskell standard library include:

Reader. This is used to pass global data or configuration information to every operation in a program. The sequential composition operation performs the left statement and the right statement, but passes an extra hidden datum to both.

Writer. Similarly, Writer allows logging or additional output from a function. Its notion of sequential composition is to perform both statements and combine their extra hidden outputs.

State. This allows stateful imperative programming to be simulated within a functional language. Each statement has an extra hidden input and output, and the sequential composition threads them together so that each statement can "see" the effect the previous one had on this hidden parametr.

List. This one is difficult to express in an imperative language: its notion of sequential composition is to perfom the second statement for each value that the left produces, thus modelling non-determinism or multiple-valued returns.

Error. Error handling in Haskell is implemented as a monad where each statement can either return a value or an error. Sequential composition runs the first statement, and then runs the second statement only if the first didn't raise an error. The result from the composed statement will an error if either statement resulted in an error, otherwise it will be the result of the second statement.

These can be layered using a technique known as *monad transformers* ###cite###. For instance, a **Reader** can be layered over any monad to give a monad which acts as the underlying monad, but where each statement can also access a global parameter.

The power of the ⟨brick⟩ compiler architecture lies in defining an "evaluator" which implements most of the semantics of the language such as symbol table management, order of evaluation issues, and the like. This evaluator is a monad transformer which will evaluate programs in *any* monad which defines a certain set of primitive operations. We can then define a number of distinct underlying monads which implement only the primitives, and yet provide a full interpreter, compiler, or typechecker.

6.2 Happy and Alex

Happy[7] is a LR parser generator for Haskell, in the style of yacc for C. It accepts input in the form of a BNF grammar. Each production of the grammar is annotated with an action, which gives a means of computing a value for that node in the derivation tree from the values of the sub-nodes. There are no restrictions on what form the values may take; any Haskell value is permissable. Common examples would be simple calculator-style grammars, where the value is simply the result of the computation, and grammars constructing abstract sytax trees where the value is a list (or other structure) of its children.

These essentially amount to an attributed translation grammar with support for synthesised attributes. Inherited attributes, where a node higher in the derivation tree passes a value to a node lower in the tree, can be emulated. Since Haskell has first-class functions, the synthesised value from a node can in fact be a function. A higher-level node can apply this function (in effect passing a value down the tree) before return its result (passing it up the tree).

Like all LR parser generators, to parse any interesting languages Happy needs a pre-processing lexical analysis stage to convert the input string into a sequence of tokens. This is provided by the Alex package[3], which is a rough equivalent of C's lex. The lexical syntax is described by a sequence of regular expressions, and Alex generates a scanner which is used to feed Happy with tokens.

6.3 LLVM

LLVM, or the Low-Level Virtual Machine, is "a compilation framework for lifelong program analysis and transformation"[9]. It provides a simple language-independent framework for compilation, and a machine-level type system.

Programs are expressed in the LLVM intermediate representation, which is essentially a typed assembly language. The LLVM type system is purely structural and tries to impose minimal constraints on the higher-level language being compiled.

LLVM provides a generic backend for compiler authors. LLVM contains a large number of standard analysis and optimisation passes defined in terms of the LLVM IR, and so can be used to perform all of the mid- and low-level optimisations that are necessary for efficient code. These include dead code elimination, partial redudancy elimination, invariant hoisting and inlining. So, the task for a compiler-writer becomes simply to generate *valid* LLVM IR and let LLVM worry about generating fast code.

There are a number of higher-level optimisations which should be applied at a level above LLVM. Most of these rely on having extra type information available. For instance, knowledge about types would allow a compiler to make extra assumptions about aliasing, and may present better opportunities for inlining. The implementation of such optimisations wouldn't generally take the form of complicated code transformations, but rather generating code in such a way that LLVM's standard optimisation passes can pick up on the extra opportunities. None of these type-based optimisations have been implemented yet, but some rough designs are noted in the ###future work section###.

6.3.1 LLVM IR

The LLVM IR is a typed assembly language for a virtual machine with:

An arbitrary number of registers. LLVM registers are simply identifiers, and any number of them can be used. All operations using IR registers are in SSA form, but LLVM contains an optimisation pass (`mem2reg`) to convert non-SSA reads and writes into SSA form.

A structural, machine-level type system. LLVM knows about the type of each operand or register, which may be an integer of any bit-width, a structure, an array, a function pointer, etc. The operations are defined in terms of the types, and so for instance there is an "instruction" to convert a pointer to a structure into a pointer to one of its fields. LLVM handles target structure layout issues and calculates offsets before generating native code.

Stack management. Stack slots are explicitly allocated and typed using the `alloca` instruction and the LLVM system keeps track of the stack offsets at which data is stored. Since the system "knows" about every access to the stack, it can safely perform optimisations such as assigning a value to a register and eliminating the stack slot, as well as producing register spill and restore code without affecting stack variables.

Calling convention support. The `call` instruction abstracts away all of the target-specific details of the platform, and so functions can be called without worrying about issues such as calling conventions, caller-save registers, stack frame management and so on.

All of these features make LLVM IR a much more pleasant target for a compiler than a normal assembly language. The LLVM typing system makes debugging code generation much easier, as many simple bugs such as accessing the incorrect field of a structure can be caught by the LLVM code-generation utilities.

Chapter 7
Extending an interpreter

Most compilers have quite a lot of seemingly redudant code. For instance, each phase of the compiler (e.g. code generation, type checking, optimisation) must "know" about the symbol table data structures. They must all understand whatever IR is being used to represent the program being compiled, and they must all implement code to process this IR, to match up operations and operands, etc.

This adds a complexity to the implementation, and makes it difficult to modify various internal compiler data structures since so many parts of the system depend upon them. This complexity makes adding new language features or modifying existing ones a significant investment of time.

One of the primary design goals of ⟨brick⟩ was the ability to quickly and easily prototype and test new features, and so flexibility and modularity of the compiler's implementation was of great importance.

7.1 Meta-circular interpreters

So, in the implementation of ⟨brick⟩, we started with the simplest form of executable definition of a language, the *meta-circular interpreter*. This is a form of interpreter where the interpreter itself is written in a high-level language (in this case, Haskell), and so many features of the language being implemented can simply be passed on to the high-level language.

For instance, the ⟨brick⟩ interpreter includes no garbage collector. Instead, ⟨brick⟩ objects are allocated as Haskell objects, and Haskell's garbage collector takes care of ensuring that they are collected and the memory reclaimed.

A similar technique was used in the implementation of closures. ⟨brick⟩ allows first-class functions, and has lexical scoping. This combination, while powerful, often leads to some implementation difficulties since the symbol table must be "closed over" when a function is returned as a value. For instance, see the following function:

```
def make_closure() do
  def x = 42
  return (function() do return x end)
end
```

`make_closure` returns a function, which itself returns `x` as defined in `make_closure`. So, when the `return` statement is executed, a closure must be created to house the function being returned. This closure must contain a reference to the code itself, as well as some representation of how the symbol table looked at the time the closure was created (so that `x` can be found, even though it is "out of scope" by the time the function is invoked).

Again, we piggybacked on Haskell's implementation of this feature: first-class functions in ⟨brick⟩ are implemented as first-class functions in Haskell. Using Haskell's closure mechanism, they close over the entire symbol table.

###code###

7.2 Monadic interpreters

Our interpreter, while being concise, is also difficult to read. This is in no small part due to the necessity of passing around the symbol table parameters (`s1`, `s2` and so on) so that names can always be resolved. We can abstract this away and hide the symbol table inside a `State` monad, so that it is always available but need not be explicitly passed around.

###code###

As an added benefit, our `eval` function is now free from the implementation details of the symbol table. The interaction between `eval` and the symbol table consists solely of a few

and we can freely substitute other more efficient representations

7.3 Generalising `eval` further

We have already generalised `eval` so that it does not depend on the concrete details of the type $\mathcal{V}$ used to represent variables. By abstracting this away, we allow different implementations of the symbol table data structures without having to change any code in `eval` to accomodate them. `eval` no longer depends on a specific data structure used to represent variables, and will now work with any type $\mathcal{V}$ as long as certain operations (`varNew`, `varGet`, `varSet` and so on) are defined on it.

We can continue applying this notion and generalise the type used to represent the value of an expression. Currently, `eval` is hardcoded to use a particular algebraic datatype which may represent a function, an integer, etc. Generalising this so that `eval` can use any type $\mathcal{E}$ on which the appropriate operations have been defined allows the same modularity as generalising variables to $\mathcal{V}$ did. For instance, a more efficient representation for values can be used without needing to modify `eval`. Alternatively, we could define new representation which kept more debugging information such as making each value keep track of the line of code which produced it. This information could be used to pinpoint the source of an incorrect value and aid in debugging. Since `eval` is generalised over the type $\mathcal{E}$, then as long as we could implement the few primitives for manipulating values in terms of $\mathcal{E}$, `eval` would transparently work on our new representation without needing to change a line of code.

As the language grows bigger, the `eval` function grows to accomodate every part of the syntax and define the semantics for the entire language. The set of primitive operations remains very small. So, by generalising `eval` over $\mathcal{V}$ and $\mathcal{E}$, we gain the ability to have multiple representations of the language's runtime data structures (optimised and debugger-friendly, for instance), without having to maintain a large amount of near-duplicated code.

So far, this is all quite standard software engineering. Reducing the number of components that see the internals of a data structure and making them instead communicate through well-defined interfaces leads to more flexible and maintainable software. In particular, it allows us to substitute alternative implementations of the data structures without needing to modify `eval`, so we can support multiple different implementations of the interpreter which share all of the same code and differ only in the primitives' implementations.

7.4 Generalising `eval` even further

The real power of this approach comes when we generalise not just over the data structures used to represent the program, but over the underlying monad. The monad is what defines the interpreter's sequencing and control flow. So far, our interpreter has been running over a `State SymbolTable` monad. This monad's notion of sequencing operations is simply to pass the state of the symbol table as defined by earlier operations into the current operation, and to keep track of any changes made so that they can be passed to future operations. The implementation of control flow (via the `cond` function) is simply to check whether a condition is true and to return either `True` or `False`, to allow `eval` to take the correct branch.

By generalising `eval` over $\mathcal{M}$, the monad used to keep track of sequencing operations in the interpreter, we open the door to many interesting parameterisations. In particular, we will see that some particular instantiations of $\mathcal{M}$, $\mathcal{V}$ and $\mathcal{E}$ will allow us to make our interpreter compile or typecheck code. We can do all this without changing the definition of the interpreter (much), just as we could allow multiple implementations of the data structure for values without needing to explicitly support them all in the `eval` function.

As shall be seen in the next chapter, this means that a compiler generating LLVM assembly code can be implemented just by implementing the primitive operations (`varGet`, `varSet`, `cond`, etc.) and defining the monad $\mathcal{M}$ and the types $\mathcal{V}$ and $\mathcal{E}$. In particular, complex flow control constructs (such as `break`, `continue`, non-local returns and the like) don't have to be implemented in the compiler at all: we just define a notion of sequencing and branching, and let the much simpler `eval` function define the meaning of the language. This frees us from the usual code-generator burden of hooking together labels and jumps, as this is done automatically according to the semantics defined by `eval`.

Chapter 8
A compiler from an interpreter

We would like to define a compiler from ⟨brick⟩ source to native code (via LLVM assembly). Not wanting to implement the large body of code necessary to emit assembly for every possible language construct, we instead define our compiler by parameterising the existing interpreter over a different monad.

8.1 A code generation monad

The compiler needs to keep track of generated code, and be able to generate fresh names for temporary variables. So, a good start for a code generation monad is:

```
type Codegen = RWS ()         -- no Reader, we're not using it
                   [LLVMInsn] -- Writer writes a list of output instructions
                   Int        -- We keep track of a single Int for fresh names
```

This is Haskell's standard Reader-Writer-State monad, where we use the Writer part to keep track of the instructions written and the State part to help us generate fresh names (Reader is unused here).

How does a compiler represent values? We can't very well return values from the primitive operations; we can't know what the answers are when compiling. Instead, a compiler represents values by symbolic names. In our case these will be LLVM register names, so we implement the type $\mathcal{E}$ simply as `String`. The primitive operation for addition takes ($\mathcal{E}$, $\mathcal{E}$) and returns $\mathcal{E}$. In the interpreter, these were two integers and it simply added them together and returned the result. In the compiler, these are two LLVM register names (represented simply as strings), so we output the code necessary to add the two operands and store the result in a fresh temporary, and then we return the name of that temporary. Again, note that we don't have to modify `eval` to implement this: `eval` blindly passes around objects of type $\mathcal{E}$ and neither knows nor cares about exactly how they are represented. Similarly, $\mathcal{V}$ is also set to `String`, this time representing a stack location, where `varGet` outputs a load instruction, `varSet` outputs a store instruction, and `varNew` allocates a new stack slot (an LLVM "alloca" instruction).

How does a compiler represent state? It doesn't have to keep track of the values of each variable as in the interpreter's symbol table, that's done by the machine's memory. All it really needs to know about is the current location of executing in the program: the compiler merely has to produce the right stream of instructions and the machine will track the rest. So, the only "program state" that the compiler must keep track of is the current position in the program. Since the compiler outputs to LLVM, the current state is represented by a single LLVM assembly label.

8.2 Representing flow control

Perhaps one of the biggest differences between an interpreter and a compiler is their treatment of flow control. An interpreter must evaluate only a single path through a program while a compiler must generate code capable of executing any possible path through the code.

In our generalised `eval`, this distinction manifests itself as the implementation of sequencing in the monad and the definition of the `cond` primitive.

`cond` in the interpreter simply checked that its argument was of the boolean data type and returned its value, either `True` or `False`. `cond` in the compiler is more complicated: we must handle both possibilities and correctly link them together in the generated code.

In the presence of conditions and branching, the generated output is not necessarily in exact order of execution. So, we will need to use the state field defined above to determine which instruction will be executed next. This leads to a simple means of sequencing code correctly: instead of generating simple instructions, we generate label-instruction-jump sequences of the form:

```
L1: instruction op1 op2 op3; goto L2;
```

Here `L1` represents the state upon control reaching the instruction, and `L2` represents the state after it is executed. If each and every instruction is output in this form, we can be guaranteed that the execution path through the compiled program will exactly correspond to the path that states were passed along in the compiler's monad. It does have the unfortunate side-effect of generating a large number of redundant sequences of the form "`goto L2; L2:` " although these are easily removed by a simple post-processing pass[8.1].

This suggests a natural mechanism for implementing `cond`: it should return *both* ***`True`*** *and* ***`False`***, each in a different "state" (i.e. assembly language label). When it returns `True` in state `L1` the "true-path" of the branch will be run and will output code starting from state `L1`. Then, when it returns `False` in state `L2` the "false-part" of the branch will be run and will output code starting from state `L2`. All that remains is for `cond` to link these together by outputting a single instruction `if condition then goto L1 else goto L2`. So, the code output will look like:

```
L0:  if condition  then goto L1 else goto L2;
L1:  true-part;    goto L3;
L2:  false-part;   goto L4;
L3:
L4:
```

Flow control thus becomes easy to implement if we allow our monad to return multiple values in multiple states. Thus, our compiler's definition of $\mathcal{M}$ becomes something like:

$$\mathcal{M} = \texttt{ListT (StateT LLVMLabel (Codegen))}$$

That is, our monad produces multiple results (`ListT`) each of which carries an LLVM label (`StateT LLVMLabel`) and produces some code (`Codegen`).

`ListT` is a monad transformer representing non-determinism. Its notion of sequencing is to run the second half of a sequenced operation for each result that the first one produces. In the compiler, this means that the code that depends on the result of `cond` will be run for each result that `cond` produces, which is the desired result.

8.2.1 Coalescing

There is an issue with this presentation so far, however. What about code like:

```
if cond1 then do
  f1()
end
if cond2 then do
  f2()
end
if cond3 then do
  f3()
end
```

8.1. Since this is implemented in Haskell, these redundant sequences don't consume a large amount of memory: they will only be computed lazily as the post-processing cleaning step demands them, so there are only a small number of them in memory at once even for a large program.

When this code is compiled, the successive calls to `cond` each cause the rest of the compiler's execution to be duplicated. Thus, four separate calls to `f3` will be emitted by the code generator, one for each path along which the code can be reached. In general, this can cause an exponential increase in code size.

We resolve this problem by introducing another primitive operation, called `coalesce`. The purpose of `coalesce` is for the `eval` function to inform the underlying monad $\mathcal{M}$ that all states reaching a given point are considered equivalent and may be combined into one.

In the compiler, `coalesce` coalesces states simply by outputting all of the labels instead of just one of them. If a set of labels `L1`, `L2`, etc. are coalesced, then the first instruction to be executed in this coalesced state will be labelled with each label from the set. This causes the execution paths to those labels to all flow to the same point in the control-flow graph, and so we can continue from there with only a single state rather than require code duplication.

In the interpreter, `coalesce` is simply a no-op: there are never multiple states to coalesce. In general, `coalesce` may be thought of as a no-op: even in the compiler it has no effect on the semantics of the compiled program, it serves simply to make the resulting output (considerably) shorter.

8.2.2 Iteration

There is, sadly, one similar case which `coalesce` is unable to address: looping.

```
while cond1 do
  f1()
end
```

If we use the existing compiler implementation and "branch both ways" on each iteration of the loop, the resulting emitted code will be infinite. Our problem is that the monad $\mathcal{M}$ has no way of "seeing" the recursion inherent in `eval`'s definition of looping. `eval` simply calls itself, and so the monad cannot spot that the next iteration of the loop will be exactly the same as the previous.

So, we introduce one more primitive operation: `fixiter`. This operation perfoms something akin to a fixpoint iteration. `fixiter` converts a monad action taking an input of type a and returning an output of type a or type b into one that takes an input of type a and returns one of type b. In Haskell syntax:

```
fixiter :: (a -> m (Either b a)) -> (a -> m b)
```

The effect is to run the monadic action, passing its output back to its input, until it returns something of type b.

By wrapping the definition of `while` withing the `eval` function in a call to `fixiter`, we can avoid the infinite-code problem. The interpreted semantics of the language do not change, as the interpreter's definition of `fixiter` is a simple recursion which runs the action over and over again until a result is produced.

The compiler, however, can now implement `fixiter` to handle loops. The input to `fixiter` will be a state, which is the result of coalescing the state as it was before the `fixiter` action was invoked, and the state as it will be when the `fixiter` action loops. Thus, we can output one sequence of code which runs in the states (i.e. from the labels) which are defined just before the while loop begins and at the end of the while loop, when it's about to loop back.

Using the yet-to-be-defined state from the end of the while loop before we've evaluated that far would seem to present a problem. Luckily, since Haskell is a lazy language, we can get away with using this state before it's been defined since all we do with the state is blindly copy it to the output.

8.3 Implementation of structures

The fundamental data type used to represent objects is the "struct": this is a mapping of string keys to string values, where the set of string keys is known at struct creation time and never changes. In the interpreter, these are implemented as a Haskell Map from string keys to values, but a more sophisticated implementation is necessary for the compiler.

Due to the nature of the type system, the amount of information we have when compiling a structure access operation varies. In some cases, we know exactly the set of keys contained in the struct (e.g. when the structure is a global constant or when we've just created it). In other cases, we know nothing about it other than that it contains the given key (e.g. when taking a function parameter with no explicit type annotations whose "x" member we access, the type-system ascertains no more than that the object does indeed have an "x" member).

In the "easy" case, where we know exactly the set of fields available, we would like to be able to compile the access down to an immediate offset. However, it is still important to keep the harder case efficient, where we don't know anything about the struct.

Structs are layed out as a series of pointer-width words.[8.2] The first word points to a "type table", which describes the layout of the rest of the structure. The "type table" must map string keys to offsets within the structure efficiently.

This problem is related to the problems of minimal perfect hashing and the Java interface lookup problem (###).

To avoid performing string comparisons, field names are hashed at compile time. To correctly handle hash collisions, each field is associated with a ### globals, unique addresses, linker, freshness ###

The actual lookup is a simple hashing scheme ### hashcodes, collisions, length-2, branches ###.

8.4 Implementation of closures

Closures are implemented on top of the struct functionality. A closure is simply a structure with a specially-named field which contains a pointer to the code to execute. If the function closes over values, those are stored as extra fields in the structure, with unique names.

If the function closes over a mutable variable, extra care must be taken to ensure that there is only one copy of the variable. For instance, consider the following code:

```
def makefunctions() do
  var v = 0
  def f1() do
    v = v + 1
    return v
  end
  def f2() do
    v = v + 1
    return v
  end
  return {func1 = f1, func2 = f2}
end
def funcs = makefunctions()
funcs.f1()
funcs.f2()
funcs.f1()
```

This returns a structure containing two fields: the functions f1 and f2 defined by makefunctions. Both of these close over the same variable v. Simply copying v into both closures is not enough: since the two functions are accessing the same variable, updates from one must be reflected in the other.

The problem is solved by not representing v as an unboxed value, but by boxing it inside a single-field structure[8.3]. Both closures refer to the same box, and all of them alias the same copy of v.

8.2. In the current implementation, there are no unboxed types so every value is represented as a pointer. In a future implementation, this constraint will be relaxed for efficiency.

8.3. Reusing the struct functionality here was a time-saving technique in implementation, a simpler box structure would have somewhat less overhead.

8.5 A typechecker from an interpreter

So far, we've shown how a single semantics for a language parameterised over a monad $\mathcal{M}$, a data type for variables $\mathcal{V}$, and a datatype for values $\mathcal{E}$ can be instantiated to give an interpreter or a compiler. Next, we show how that can be generalised to form a type-checker.

The algorithms for analysing the constraint graph of a program have been described at length in ###part I. What remains is the initial constraint generation pass: we must be able to construct such a constraint graph from an arbitrary input program.

We'd rather not define this pass in terms of the concrete syntax of the language or in terms of symbols and symbol tables, since we seem to have already implemented that code for the compiler and interpreter and would like to avoid duplicating it. Instead, we will attempt to re-use our generic `eval` code by finding an implementation of the primitives and a definition of $\mathcal{M}$, $\mathcal{V}$ and $\mathcal{E}$ which causes the result of our interpreter to be a constraint graph.

The approach is not dissimilar to that of abstract interpretation. Indeed, all of the parameterisations of the `eval` function may be considered abstractions of the interpreter. In particular, types may be considered abstractions of the sets of values which they represent, and so the types manipulated by the typechecker can be considered an abstraction of the values manipulated by the interpreter.

For a detailed treatment of this relationship between type inference and abstract interpretation, see Cousot's work in [?].

In this implementation, we describe the constraint generation process by instantiating $\mathcal{M}$ to be a monad very similar to the one used for the compiler. Instead of the bottom-most monad generating a list of instructions, we have it generate a set of constraints. Also, we can ignore the state parameter used in the compiler's $\mathcal{M}$, since the types of terms are required to be independant of the point in the execution of the program. Since there is no state parameter, coalescing is easy to implement, and since we allow recursive constraints, `fixiter` is relatively easy.

The types $\mathcal{E}$ and $\mathcal{V}$ simply represent type variables. Each of the actual operations of the language which operates upon values can be expressed as a constraint upon type variables. For instance, the primitve operation `apply` (used to apply a function to its argument) takes a pair of $\mathcal{E}$ (function and argument) and returns a single $\mathcal{E}$. It can be implemented in the typechecker as taking (a, b) and returning a fresh variable c while building the constraint $a \leqslant b \rightarrow c$. Each of the primitive operations can be defined in this way, giving us a type checker which builds constraints as an abstraction of the operation of the interpreter.

Thus, with a single definition of `eval`, as well as getting a compiler we also gain a type-checker.

Chapter 9
Future work

Chapter 10
Conclusions

Appendix A

BNF grammar for the syntax of ⟨brick⟩

Appendix B
Detailed typing rules for ⟨brick⟩

Subjred as abstract interpretation a la Cousot, fits with compiler implementation, easy to verify, don't have to throw it away each change[?]

Bibliography

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, 1974.

[2] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Trans. Program. Lang. Syst.*, 15(4):575–631, 1993.

[3] C. Dornan, I. Jones, and S. Marlow. Alex: A lexical analyser generator for Haskell. *University of Glasgow*, 1995.

[4] G. Dubochet and M. Odersky. Compiling structural types on the JVM: a comparison of reflective and generative techniques from Scala's perspective. pages 34–41, 2009.

[5] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Type Inference for Recursively Constrained Types and its Application to OOP. *Electronic Notes in Theoretical Computer Science*, 1:132 – 153, 1995. MFPS XI, Mathematical Foundations of Programming Semantics, Eleventh Annual Conference.

[6] J. Gil and I. Maman. Whiteoak: introducing structural typing into java. *ACM SIGPLAN Notices*, 43(10):73–90, 2008.

[7] A. Gill and S. Marlow. Happy: The parser generator for Haskell. *University of Glasgow*, 1995.

[8] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient recursive subtyping. pages 419–428, 1993.

[9] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. Mar 2004.

[10] X. Leroy. Polymorphism by name for references and continuations. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 220–231. ACM, 1993.

[11] D. Malayeri and J. Aldrich. Integrating nominal and structural subtyping. *ECOOP 2008–Object-Oriented Programming*, pages 260–284, 2008.

[12] P. Morris, T. Altenkirch, and C. McBride. Exploring the regular tree types. *Types for Proofs and Programs*, pages 252–267, 2006.

[13] François Pottier. Type inference in the presence of subtyping: from theory to practice. Research Report RR-3483, INRIA, 1998.

[14] François Pottier. A framework for type inference with subtyping. *SIGPLAN Not.*, 34(1):228–238, 1999.

[15] M. Tofte. Type inference for polymorphic references. *Information and computation*, 89(1):1–34, 1990.

[16] Valery Trifonov and Scott Smith. Subtyping constrained types. 1145:349–365, 1996. 10.1007/3-540-61739-$6_5$2.

[17] A.K. Wright. Polymorphism for imperative languages without imperative types. 1993.

[18] A.K. Wright. Simple imperative polymorphism. *Lisp and symbolic computation*, 8(4):343–355, 1995.