# Declaration

I hereby declare that this thesis is entirely my own work and that it has not been submitted as an exercise for a degree at any other university.

_____ April 11, 2011

Stephen Dolan

# Permission to Lend

I agree that the Library and other agents of the College may lend or copy this thesis upon request.

_____ April 11, 2011

Stephen Dolan

# Acknowledgements

**Abstract**

We present Brick, a new imperative programming language. Brick features static typing with full type inference, subtyping, and object-oriented features. The type inference engine is based heavily on previous work, but we present some new implementation techniques and a novel method of integrating nominative classes with inference.

Brick also has an interesting modular implementation which abstracts over concepts like control flow and naming, removing redundancy present in traditional langauge implementations. This enables a single implementation to be re-used for interpretation, compilation and type checking.

# Table of contents

# Chapter 1
# Introduction

*They have been at a great feast of languages, and stolen the scraps.*
***William Shakespeare,***
***"Love's Labour's Lost'***

This report describes the design and implementation of the object-oriented imperative language Brick. Brick has a sophisticated type system, supporting global type inference, subtyping and seamless integration of nominative and structural typing. Brick's prototype implementation has an unusual and interesting compiler architecture, combining and abstracting several previously separate parts of a language implementation.

## 1.1 Background

The design of the language sought to improve upon most imperative, object-oriented languages by importing a number of advanced features from functional programming languages. Features like type inference and structural typing are common in statically-typed functional languages, and seemed worth trying out in an object-oriented imperative setting.

### 1.1.1 Type systems

The type systems of contemporary programming languages may be classified along a number of axes:

**Static versus Dynamic.** Static type systems perform their type analysis and determine whether a program is type-correct before the program runs at all, while dynamic type systems check for errors during execution. The important distinction is that a static type system can guarantee the absence of type errors from any possible run of a program; a dynamic type system can only check for the absence of type errors in a given execution of a program.

**Manifest versus Inferred.** In the case of static typing, how many type annotations must be provided by the programmer? Manifest type systems are easier to implement, since the problem of verifying existing type annotations is easier than the problem of producing them. Type inference places less of a burden on the programmer, and often increases the signal-to-noise ratio of the program text by removing irrelevant annotations.

**Strong versus Weak.** These terms have fuzzy definitions, largely because "weak" is considered perjorative and every language designer wants to call their system "strongly typed". To a first approximation, these terms indicate how the system responds to seemingly ill-typed expressions: will an attempt to add a string to an integer be an error ("strong typing", as in Java, C#, Python, etc.) or will it cause a silent conversion ("weak typing", as in PHP, Perl, etc.).

**Nominative versus Structural.** Are types considered compatible based on whether they support the same set of operations, or based on whether they declare the same type? Are two differently-named types interchangeable if they have the same definition? Many "scripting" languages (Python, PHP, Ruby) prefer structural typing, whereas others (Java, C#) use only nominative types.

By these metrics, BRICK is strongly and statically typed, with inference: typechecking is performed without reference to a particular set of inputs, there are no implicit conversions between unrelated types and types need not be explicitly spelled out. Where BRICK falls on the nominative-structural scale is a more interesting question, which will be answered thoroughly in section 4.5.

### 1.1.2 Subtyping and object-orientation

Subtyping allows an object with a specific interface to be used when a general one is expected.

Subtyping is one of the primary mechanisms for producing modularity in object-oriented programs: if a method takes a parameter with certain interface, then any object which provides that interface may be passed. The method doesn't care about which other operations the parameter object may support; it uses only a specific subset of them. More modular programs can be written as the method need not know about the exact details of what else the object can do.

Subtyping allows information to flow freely from specific to general. No tedious "wrapping" or "unwrapping" is required to pass the object with a detailed interface to the method that required only part of the interface, since subtyping allows one to transparently "forget" about parts of an interface.

Treatments of subtyping require careful thought about the direction of data-flow: information is only allowed to flow from a specific type to a general type. So, when providing input to a module we may give it an object of a more specific type than required, but when receiving output we may treat the output as a more *general* type than provided.

Subtyping presents interesting problems for type inference, as a system of subtyping constraints is more difficult to solve than a system of term equations. However, the typings produced by an inference system with subtyping are more precise as they take into account the direction of data flow.

### 1.1.3 Nominative and structural types

Subtyping systems can be divided into "nominative" and "structural". Nominative systems consider types to compatible only if they are explicitly declared so, while structural systems consider compatible any types that have a similar interface.

For instance,

Java

```
void writelist<T>(Iterable<T> list,
                  Writer file) {
  foreach (T i: list) {
    file.write(i.toString());
  }
}
```

Python

```
def writelist(list, file):
  for i in list:
    file.write(str(i))
```

The two examples implement the same function, once in Java and once in Python. The Java version requires that its first argument is declared as implementing the interface `Iterable<T>`, while the Python version will accept any list-like object which has certain methods.

Nominative subtyping has historically been standard in mainsteam statically-typed OO languages like Java or C++, where all relationships between types are explicitly[1.1] indicated. Structural typing is more common in research languages, as structural type systems can be defined which accept precisely the set of programs whose execution would not cause an error.

Both systems have advantages: nominative typing recognises only those relationships defined by the programmer, and hence has a smaller chance of accepting programs which are semantically wrong but do not cause an execution error. Structural subtyping sees all possible compatibilities, and so requires less boilerplate annotations to reuse common code.

1.1. and verbosely!

### 1.1.4 "Polymorphism"

"Polymorphism" in general means using the same code with 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 `Square`s are `Shape`s. 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`$[a] \to a$ for all $a$. Subtype polymorphism is implicit in the typing rules, and the shape area example above would have type `Shape` $\to$ `float` (which can transparently be used as if it had type `Square` $\to$ `float`).

## 1.2 Outline of syntax

Brick's syntax is broadly similar to that of Lua or to a lesser extent Ruby. A function definition might look like:

```
def myfunction(argument) do
  # Comments are introduced with the '#' symbol
  var x = 42, y = 16   # Variables are declared with ``var''
  def pi = 3.14159     # ``def'' is like ``var'', but for constants
  if x > 0 then do     # if-then-else for conditionals
    something()        # function call
  else do
    while x <= 0 do    # while loops
      something(       # first class functions and closures
        function (x) do
          print(x)
        end)
    end
  end
  var pt = {x=10,      # structure types
            y=20}
  var ptx:int = pt.x   # type annotations and structure access
  return pt.y          # return values
end
```

Specific features of Brick will be described in the following chapters, and a BNF grammar is given in appendix A.1.

## 1.3 Brick's type system

Brick has a rich type system, supporting structures, objects, first-class functions and classes. It also supports global type inference, so type annotations are not necessary (although may be included as a form of machine-checked documentation).

### 1.3.1 Constructing types

BRICK supports a number of primitive types, such as `bool` which represents Boolean values and `int` which represents integers. There are also more complex types, such as `int` → `int`, which represents the type of functions taking an integer and returning an integer, and {`x`: `int`, `y`: `int`}, which represents the type of structures having an `x` field and a `y` field, both of integer type.

These basic types are treated uniformly as *type constructors*. Each type constructor takes a number of types as parameters, and constructs a more complex type out of them. For instance, `int` and `bool` are type constructors taking no parameters, while → is a type constructor taking two parameters: the argument and result types. Similarly, {`x`, `y`} is a type constructor taking two parameters: the types of the `x` and `y` fields.

### 1.3.2 Constraints and subtyping

A program is typechecked by constructing a series of subtyping constraints on its type. For instance, this simple function is of type $f$:

```
def f(x) do
  var y = x.part
  return y.subpart
end
```

$$\begin{array}{rcl} a \to b & \leqslant & f \\ a & \leqslant & \{\texttt{part}: c\} \\ c & \leqslant & \{\texttt{subpart}: b\} \end{array}$$

The program will run without type errors if the constraints on the right are satisfied. The constraints are resolved by the type inference engine, which eventually pronounces the program type-correct or gives an error.

These subtyping constraints require a notion of subtyping between types. As described previously, subtyping works in a different direction for inputs and for outputs. A component providing an output may be considered *a fortiori* to provide any less specific type of output; thus the output type becomes more specific as the component type becomes more specific. An input to the component, on the other hand, may be given any more specific type of input; thus the input type becomes more specific as the component type becomes more *general*.

For instance, a process taking as input any box and producing a wrapped box can be safely considered as a process taking blue boxes and producing wrapped boxes. Thus,

$$\text{box} \to \text{wrapped box} \leqslant \text{blue box} \to \text{wrapped box}$$

It may not, however, be considered to be a process producing blue boxes, as not all wrapped boxes need be blue. But, it may be considered to produce boxes in general, as all wrapped boxes are boxes. Thus,

$$\text{box} \to \text{wrapped box} \leqslant \text{box} \to \text{box}$$

This "direction of subtyping" is a property of each parameter of the type constructor. Those parameters which work like outputs are said to be *covariant*: the type gets narrower as the parameter gets narrower. Conversely, input-like parameters are said to be *contravariant*.

A subtlety arises with parameters can which can be used as both inputs and outputs, such as the fields of a structure (which can be read or modified). To see how this is resolved, see section 4.1.1.

### 1.3.3 Integrating nominative and structural typing

The language seeks to combine nominative and structural typing seamlessly, and allow a programmer to mix-and-match the two styles.

BRICK is not the first language to seek to combine nominative and structural typing for objects. Whiteoak[11] 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[7] 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[20] 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.

### 1.3.4 The type system and inference engine

The description of the type system and inference engine will be split into two sections. The next two chapters will describe the inference engine abstractly, without reference to a specific set of type constructors (although $\rightarrow$ and structure types will be used to illustrate examples). The abstract inference engine will work for any system of type constructors which satisfies certain properties, formalised in 2.5.1. The following chapter will describe Brick's parameterisation of this generic system, and how it encodes functions, classes and objects, and how it combines the notions of structural and nominative types.

The system is based originally on work defining a type system with subtyping and recursive types for the $\lambda$-calculus by Amadio and Cardelli[2], with further exploration in [17, 22]. The notion of a "constrained type", to represent the type of a complex polymorphic function, was explored by Palsberg et al. in [23, 24, ?] and its subtyping problem by Smith and Trifonov in [33]. The application of these ideas to object-oriented programming appeared in Eifrig et al. in [8, 9]. The type inference engine and constraint solver described here is heavily based upon the one described by Pottier in [29, 28, 27].

Most of the statements about these type systems are presented here without proof. The interested reader may check those references for a more in-depth discussion.

## 1.4 Language implementation

The implementation sought to reduce the workload associated with the development of a compiler through modularity. In most language implementations, a compiler, interpreter and type-checker are largely separate bodies of code. Many algorithms which on the surface appear similar (such as analysing data flow through the program) must be implemented separately for each component.

Brick's implementation, using functional programming techniques, abstracts away the common parts of the language's semantics into a single "evaluator". This is used by all phases of the compiler, and contains a single definition of the common semantics (e.g. control flow, symbol resolution, bindings, matching operands to operators) while each of the phases need only implement a small set of primitive operations (e.g. loads, stores and functions).

Thus, we gain a compiler and a type-checker, without having to write them! We write a very simple interpreter, and re-use its definition of the language in the two other phases.

In particular, this reduces the amount of work necessary to experiment with a new language feature: it is much easier to add a feature to this system than to a traditional compiler, which is very valuable in an exploratory project such as this one. When a language feature requiring a new primitive operation is added, it can first be implemented in the interpreter and then most of that implementation can be re-used if it is decided to add support to the compiler.

For instance, those parts of the implementation which deal with issues such as high-level control flow constructs or resolving symbol names are only implemented once, in an abstract fashion. This abstract implementation can be transparently re-used by the various components of the implementation.###

# Part I: Language Design

*a sea of Greek letters*

# Chapter 2
# Type inference as constraint solving

*The more constraints one imposes, the more one frees oneself of the chains that shackle the spirit ... the arbitrariness of the constraint only serves to obtain precision of execution.*

***Igor Stravinsky,***
***"The Poetics of Music"***

## 2.1 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:

- Primitive types such as `int` and `bool`
- Structure types such as $\{\texttt{foo}: a, \texttt{bar}: b\}$, 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 classes (see section 4.3)
- 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 chapter 4, 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.

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 chapter 4, particularly section 4.5.3).

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 section 2.5.1.

### 2.1.1 $\top$ 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.

## 2.2 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 tree are non-nullary type constructors (such as $\rightarrow$ or $\{$`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 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 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.

**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 addition 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, 17, 22, 28].

### 2.2.1 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[2.1]. 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 $\mu$ 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. 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 types 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[28].

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

### 2.2.2 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

$$\begin{aligned} g &= c(l_1{:}\, t_1, l_2{:}\, t_2, ...) \\ g' &= c'(l'_1{:}\, t'_1, l'_2{:}\, t'_2, ...) \end{aligned}$$

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 $t' \leqslant t$, when contravariant) for all types associated to a label that $c$ and $c'$ have in common.

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 [17] 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[28]. 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.

## 2.3 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. Ground types correspond roughly to the monomorphic types of functional programming languages such as ML and Haskell.

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 {`name`: `string`}.

A first attempt at typing such a function might assign it the type {`name`: `string`} → {`name`: `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 {`name`: `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 [9]). The type of our "logging identity" function then becomes

$$a \to a \backslash a \leqslant \{\texttt{name}{:}\,\texttt{string}\}$$

The $\backslash$, read as "where", separates the type from a series of constraints on the variables defined in the type. In general, we'll write rc types as $a \backslash C$ where $a$ is a type or type variable and $C$ is a set of constraints.

### 2.3.1 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.

The set of ground types denoted by an rc type $a \backslash C$ is the set of ground types that can be formed by substituting the free variables of $a \backslash C$ with ground types, as long as the constraints in $C$ still hold. This set is 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.

## 2.4 Constraints and well-typedness

The process of type inference is 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).

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.

## 2.5 Structural decomposition

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 $L'$. Let $l_1$, $l_2$, ... denote arbitrary elements of $L$ (and $l'_1$,$l'_2$,... denote elements of $L'$)

We have the constraint

$$c(l_1\colon \tau_1, l_2\colon \tau_2, ...) \;\leqslant\; c'(l'_1\colon \tau'_1, l'_2\colon \tau'_2, ...)$$

For each $l \in L \cap L'$, let $\tau$ be the type assigned to it on the left-hand side of the constraint and $\tau'$ 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 $\{\texttt{field1}\colon a, \texttt{field2}\colon b\} \leqslant \{\texttt{field1}\colon c\}$ 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, 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 decomposition operation will be written $\text{subc}(\tau \leqslant \tau')$. It is only well defined when $c \leqslant c'$, where $c$ and $c'$ are the constructors of $\tau$ and $\tau'$ respectively.

### 2.5.1 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.

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 $\mathbb{C} \to \mathcal{P}(\mathbb{L})$, called the arity of a constructor
- A mapping $\mathbb{L} \to \{+, -\}$, 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 $c_1 \leqslant c_2 \leqslant c_3$, $\text{arity}(c_1) \cap \text{arity}(c_3) \subseteq \text{arity}(c_2)$

The second condition here is known as *convexity of arity*, and is important to the workings of "subc" above. "subc" operates only on the common parameters of the two constructors, and so this condition 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.

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 $\{+, -\}$ 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 4.1.1).

## 2.6 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 $\tau, \tau'$ such that $(\tau \leqslant \tau') \in C$ then $\text{subc}(\tau \leqslant \tau') \subseteq C$

Any closed graph is satisfiable, and any satisfiable constraint set can be closed[28, 8]. So, satisfiability can be checked by attempting to compute the closure of the constraint set.

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 3
# The type inference engine

*It is not really difficult to construct a series of inferences, each dependent upon its predecessor and each simple in itself. If, after doing so, one simply knocks out all the central inferences and presents one's audience with the starting-point and the conclusion, one may produce a startling, though perhaps a meretricious, effect.*

***Sir Arthur Conan Doyle,***
***"The Dancing Men"***

In this chapter, we'll describe how the constraints representing a program are resolved to yield a type or a type error. We also describe how the constraint set is *optimised*: a constraint set describing an entire program can quickly become unwieldy, and techniques are needed to reduce it to a manageable size.

Two invariants are proposed, both from Pottier[28, 29]: the small terms invariant, which requires that each constraint have only a single level of structure (similar to three-address code in a compiler), and the mono-polarity invariant, which allows us to distinguish between variables used as upper and lower bounds.

## 3.1 The small terms invariant

When representing types inside the inference engine, we use Pottier's "small-terms invariant". That is, we avoid representing complex nested types with multiple layers of constructors and instead ensure 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 sub-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 $y = c \to d$.

We[3.1] define a *small constructed type* as one that is of the form $c(p, q, r, ...)$ where $c$ is a type constructor and its parameters $p, q, r, ...$ 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 $\{x \leqslant a \to b, a \to b \leqslant x\}$. As it happens, thanks to the garbage collection algorithm described below, we can in fact always drop one of these.

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 (section 3.6.2).

3.1. Provisionally, see the next section

## 3.2 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 $a$:

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

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.

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

These operations for combining bounds are the lattice-theoretic greatest 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[3.2]. 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.

## 3.3 The mono-polarity invariant and garbage collection

Before we can describe the optimised representation of constraint sets, we must draw a distinction between *positive* and *negative* type variables and constructed types.

The type inferred for a particular term is a lower bound: the term may be used with any supertype of the inferred type. We say that the type "appears positively". If a constructed type appears positively, then its parameters appear positively when covariant and negatively when contravariant (and vice versa for negative appearances). If a variable appears positively, then the left-hand side of any constraint whose right-hand side consists of that variable appears positively, and similarly for negative appearances. For example, if the type of a term is $a$, with the constraint graph:

$$\begin{aligned} b \to c &\leqslant a \\ d \to e &\leqslant b \\ c &\leqslant f \end{aligned}$$

$b \to c$ appears positively, as a lower bound of $a$ (assumed positive, as it is the root of the type). Then, decomposing this constructed type, $c$ appears positively and $b$ appears negatively. Following on, $e$ appears positively and $d$ appears negatively, while $f$ appears negatively.

The mono-polarity invariant is this: *no variable appears both positively and negatively.*

This is quite simply enforced: if we have a constraint set in which we would like to have a variable appear with both polarities, we can simply remove the offending variable and replace it with a pair of variables, one positive and one negative, with a constraint between them.

3.2. For a more thorough presentation of the $\sqcup$ and $\sqcap$ operators (including their generalisation to non-small terms), see [28].

### 3.3.1 Small constructed types

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 $(a^-, b^-, ...)$[3.3].

Likewise, the *small constructed terms* are divided into two classes. Positive small constructed terms will be written $\tau_1^+$, $\tau_2^+$, ... while negative ones are $\tau_1^-$, $\tau_2^-$. We will occasionally abstract over the polarity of a term or variable and write $a^v$ or $\tau^v$ to mean a term of polarity $v \in \{+, -\}$. Similarly, $\square^v$ will be used to abstract over $\sqcup$ and $\sqcap$, where $\square^+ = \sqcup$ and $\square^- = \sqcap$.

A small constructed term is defined as:

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

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 types to merged via $\sqcup$ into a single small constructed term, while any set of negative small constructed types can be merged via $\sqcap$.

### 3.3.2 Garbage collection of constraint sets

If a variable appears neither positively or negatively in the constraint graph, it may be removed. This is the principle of *garbage collection*, described fully in [33, 28]. This is only tangentially related to garbage collection of heap values in a running program: it refers to the removal of redundant parts of a constraint set.

For instance, suppose we have the rc type:

$$a \to b \backslash a \leqslant b, b \leqslant d \to e$$

Since the type $a \to b$ appears positively, $b$ is positive. $d \to e$ appears only as an upper bound for a positive variable, and hence $d$ and $e$ appear neither positively nor negatively. As shown by Smith and Trifonov in [33] and Pottier in [28], we can safely remove this constraint.

As well as this, we can safely remove any constraints between variables if one is not reachable. In general, the only constraints which we need to actually keep in our constraint graph are ones of the form:

$$\begin{aligned} \text{positive constructed type} &\leqslant \text{positive variable} \\ \text{negative variable} &\leqslant \text{negative constructed type} \\ \text{negative variable} &\leqslant \text{positive variable} \end{aligned}$$

This allows us to define an optimised constraint graph representation and incremental closure algorithm which avoid storing the data which would be removed by the garbage collection algorithm.

## 3.4 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 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:

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

3.3. The polarity signs are considered part of the name, there is no implied relationship between $a^+$ and $a^-$.

In the notation of [28, 29] $\Omega^\tau(a^+) = \tau^+$ would be represented as $C^\uparrow(a^+) = \tau^+$, while in [33] it would be $\tau^+ \leqslant a^+ \in K$. The advantage of our 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. Thus, we only store lower bounds for positive variables, and only upper bounds for negative variables, thus reducing the storage requirements of type inferencing over previous work in this area.

### 3.4.1 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[26], 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's garbage collector.

## 3.5 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 defined in section 2.6). 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 $\Omega^\tau$ 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 [28] and [8], adapted to our constraint graph representation.

```
addConstraint(a⁺,b⁻):
  for each c⁻ ∈ Ω^V(a⁺), d⁺ ∈ Ω^V(b⁻):
     add c⁻ to Ω^V(d⁺)
     add d⁺ to Ω^V(c⁻)
     set Ω^τ(a⁺) := Ω^τ(a⁺) ⊔ Ω^τ(d⁺)
     set Ω^τ(b⁻) := Ω^τ(b⁻) ⊓ Ω^τ(c⁻)

  for each constraint e⁺ ⩽ f⁻ ∈ subc(Ω^τ(a⁺) ⩽ Ω^τ(b⁻)):
     addConstraint(e⁺, f⁻)
```

The invocation of "subc" above is not well-defined when the constructor of $\Omega^\tau(a^+)$ is not a subtype of the constructor of $\Omega^\tau(b^-)$ (see section 2.5). If this occurs, the constraint is unsatisfiable, and so closure fails giving a type error.

## 3.6 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 unwieldy and slow to manipulate, such large types are 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 [27, 16]. Many of these (such as the removal of cycles in [27]) are in fact unnecessary in this system since they are less powerful than 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 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 [33] defined canonisation, and Pottier [28] defined canonisation and minimisation. Our implementation is heavily based on Pottier's: the algorithms are fundamentally identical, but their description is marginally different as our constraint graphs have a simpler form.

In [28], Pottier's garbage collection does not necessarily produce a closed output constraint graph given a closed input. He then defines a weaker closure-like property 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 [28] as "perfect", and hence the results therein about the validity of the various optimisations (garbage collection, canonisation and minimisation) still hold for this system.

### 3.6.1 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 merged variables with more than two elements appearing as a parameter to a constructed term may be removed and replaced with the singleton set $\{S^v\}$, thus removing all instances of $\sqcap$ and $\sqcup$ from the graph.

Of course, the actual implementation of the 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 other 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 have 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:

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

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 $(e^+ \sqcup g^+) \to (f^- \sqcap h^-)$. The algorithm then introduces variables $y^+$ and $z^-$ to stand for $e^+ \sqcup g^+$ and $f^- \sqcap h^-$. Thus, the final constraint graph becomes:

$$\begin{aligned} a^- \to b^+ &\leqslant \texttt{f}^+ \\ a^- &\leqslant \{\texttt{doSomething}: x^-\} \\ () &\leqslant b^+ \\ x^- &\leqslant y^+ \to z^- \\ \texttt{int} &\leqslant y^+ \end{aligned}$$

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

### 3.6.2 Minimisation

The minimisation algorithm in [28] grew out of other work by the same author [27] 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 $\sigma$ if, for all variables $a, b$ in $\sigma$ such that $a \equiv b$:

- $a$ and $b$ have the same polarity
- $\Omega^\tau(a) \equiv \Omega^\tau(b)$ where equivalence of small canonical constructed terms means they have the same constructor and the parameters are componentwise equivalent
- $\Omega^\mathcal{V}(a) = \Omega^\mathcal{V}(b)$

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[3.4]:

```
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 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 inference engine generates a type with many redundancies for this function. The actual type generated, after canonisation but before minimisation, is:

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

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 3.8, this type would be rendered simply as $a \to a \setminus a \leqslant \{\texttt{next}\colon a\}$.

## 3.7 rc type subsumption

Occasionally the question arises of whether one rc type is a subtype of another. This is not needed for most type inferencing operations, since the closure algorithm checks solvability of constraint sets without needing to know this. 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.

3.4. The condition on the "if" is simply `true` to avoid introducing boolean types; the types are independent of the execution path.

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.

### 3.7.1 Subsumption

In order to check user type annotations for polymorphic defitions such as functions, we need to be able to calculate whether one type with free variables (the inferred type) is a subtype of another type with free variables (the declared type).

This is the subsumption relation $\leqslant^{\forall}$: $a'\backslash C' \leqslant^{\forall} a\backslash C$ if all terms of rc type $a'\backslash C'$ are also of rc type $a\backslash C$. Equivalently, $a'\backslash C' \leqslant^{\forall} a\backslash C$ if the denotation[3.5] of $a'\backslash C'$ is a subset of the denotation of $a\backslash C$. No efficient algorithm for calculating this is known, but efficiently decidable approximations have been defined[28, 33]. The algorithm answers queries of the form $a^+ \leqslant b^+$ (or $c^- \leqslant d^-$). Essentially, the question being answered by the algorithm is if the (disjoint) constraint sets of $a$ and $b$ were combined, would the resulting constraint graph be more restrictive than that of $b$ on its own?

This question is answered using the following algorithm: we perform the incremental closure algorithm to combine the constraint sets. However, when we are to add a constraint purely defined on the variables in $b$'s graph (or $c$'s graph, in the negative case), then rather than add it to the graph by incremental closure, we use the *entailment* algorithm to prove that it doesn't need to be added. Essentially, entailment tells us that a constraint is "already present" in the constraint graph, or that it is implied by what is already there.

If this succeeds, it implies that we can combine $a$'s and $b$'s constraint graphs and end up with a constraint graph no more restrictive than that of $b$, and hence $a \leqslant^{\forall} b$.

### 3.7.2 Entailment

A constraint graph $C$ entails a constraint $x^- \leqslant y^+$ if adding that constraint to $C$ would not reduce the set of solutions. That is, $C \vdash x^- \leqslant y^+$ 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. For instance, if we have the constraint set $\{a \leqslant \bot \to a, \bot \to b \leqslant b\}$, then in any solution $a \leqslant b$, even though this constraint is not present in the set.

Like subsumption, entailment is not known to be efficiently decidable. However, there are a number of sound, but incomplete entailment algorithms. A very simple one would be to check whether the constraint was literally present in the constraint set, but this would fail to prove too many entailments to be useful.

Instead, we use a powerful approximation to entailment based on an axiomatisation of provable entailment from [28]. The algorithm will answer queries of the form $a^+ \leqslant b^-$ and will return a boolean result. If the algorithm returns true, the constraint is entailed by the constraint graph and adding it to the graph would not shrink the set of solutions.

```
HIST := ∅
entailed(a⁺,b⁻) =
  if (a⁺ ⩽ b⁻) ∈ HIST or a⁺ ∈ Ω^V(b⁻) then
    True
  else
    set HIST := HIST ∪ {a⁺ ⩽ b⁻}
    if entailed(p) for p ∈ subc(Ω^τ(a⁺) ⩽ Ω^τ(b⁻)) then
      True
    else
      False
```

3.5. see section 2.3.1

`HIST` denotes the set of entailments previously proven or previously added as hypotheses in the current proof. As explained in [28], it is valid to add hypotheses to `HIST` before the hypothesis is fully proven as proving an entailment recursively in terms of such hypotheses demonstrates the existence of an infinite regular proof.

The algorithm is similar to the algorithm in [17] for efficiently deciding the subtype relation between two ground types expressed as regular trees.

## 3.8 Display

The representation of constraint graphs in the typechecker, while efficient, is almost entirely incomprehensible to a human programmer. In particular, the small-terms invariant results in a constraint graph which contains 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 $d^+ \to e^-$ 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 separate type variables for 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 $a \to a$.

Both of these type graphs can be made readable by applying a simple substitution: replacing type variables with their unique bounds. That is, if a variable $a$ has an empty set of variable bounds $\Omega^{\mathcal{V}}(a)$, then it is safe to replace $a$ with its constructed bound $\Omega^{\tau}(a)$. Similarly, if the constructed bound does not constrain $a$ (that is, $\Omega^{\tau}(a) = \bot$ if $a$ is positive, or $\top$ if $a$ is negative), and $a$ has only a single variable bound, $a$ can safely be replaced with that variable.

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

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

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

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

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

$$\texttt{f2}^+ \;\mapsto\; a^- \to b^+ \qquad\qquad b^+ \;\mapsto\; a^-$$

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

$$a \to a$$

# Chapter 4
# Semantics and object model

*I paint objects as I think them, not as I see them.*
***Pablo Picasso***

In this chapter, the main features of the Brick language and how they integrate with the type inference system will be discussed, including the interaction of structures and classes and how polymorphic functions are handled.

## 4.1 Structures

Structures are a mapping from field names to values. The set of fields of a structure is immutable after the structure is created, but the values can be read and written.

The subtyping rule for structures is that a structure type $S_1$ is a subtype of $S_2$ if $S_1$ has a superset of the fields and the field types are compatible. Exactly what is meant by "compatible" will be deferred for a moment, and answered in the section describing how mutability is integrated into the type system.

Structures are created with a syntax like $\{\mathtt{x} = 42, \mathtt{y} = 17\}$. Fields are accessed as `s.field` and assigned as `s.field = val`. The subtyping rule means you can safely pass a structure to a function expecting a structure with arbitrary subset of the fields present. This is vital to correctly inferring types for bodies of code which use structures, as it means that a function need not reference all of the fields to be compatible with a structure.

### 4.1.1 Mutability and typing

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 `Dog`s to a function expecting an array of `Mammal`s. 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 `Dog`s cannot be considered to be an array of `Mammal`s 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`). To solve this problem, we use an elegant trick described in [28], attributed to Luca Cardelli, and Smith and Trifonov (independantly): we introduce *two* type parameters, one contravariant and one covariant.

Thus, in the array above, the argument to `f1` will have type array($\bot$, `Mammal`). The contravariant type is unconstrained, and therefore remains as $\bot$. The array being passed, array(`Dog`, `Dog`), is a subtype of this type and thus correct to pass. The argument to `f2` will have type array(`Mammal`, $\top$) since it only uses the parameter contravariantly. This is not a subtype of array(`Dog`, `Dog`), so the function is correctly rejected. array(`Animal`, `Animal`) is, however, so it would be type-correct (and make sense!) to pass such an array to `f2`.

This technique of typing mutable objects with a pair of type variables is used to type structure fields as well. So, a fully-specified structure type might look like {`f1`: $a/b$}, where $a$ is the (contravariant) type used for assignments to the field, while $b$ is the (covariant) type used for reading from it. The type system will ensure that $a \leqslant b$, so anything written can be read back out.

## 4.2 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. 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 had type $t'$, where $t \leqslant t'$. This could be done to clarify the programmer's intentions and to indicate the function of the code to a user.

As an example, consider a program which uses the type {`x`: `float`, `y`: `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[4.1]:

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

This will be given the type {`x`: $a$} $\to a$ by inference. However, the programmer wishes to indicate that this function is to be used only with points, and not with just 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 {`x`: `float`, `y`: `float`} $\to$ `float`, which better indicates the programmer's intentions. Since {`x`: `float`, `y`: `float`} $\to$ `float` is a supertype of {`x`: $a$} $\to a$, the annotation is valid.

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

### 4.2.1 Checking type annotations

To verify that a type annotation is correct, we must check that the declared type of a term is some supertype of the inferred type. That is, the type the programmer uses to declare a term must be equal or less specific than the type inferred from the body: he cannot claim the code does something it can't.

Annotations for generalised and ungeneralised terms are checked somewhat differently, since the former may contain free variables and is therefore more complicated. Ungeneralised terms are checked using the entailment algorithm: if the programmer writes `x:int`, we check that adding the constraint $x \leqslant$ `int` does not restrict the set of solutions (alternatively, that this constraint is implied by the constraints inferred).

If an annotation is placed on a generalised term, for instance if the programmer defines an identity function `id` and writes `id:a => a forall a`, we must use the subsumption algorithm in order to cover all of the possible instantiations of the free variable $a$.

## 4.3 Classes

Classes are built atop the structure functionality: each class is a subtype of the structure type with the same set of members. This facilitates the integration of nominative and structural types: an object of nominative type (an instance of a class) can be passed to a function expecting a structural type, and the subtyping rules ensure that it will work. There are many subtleties with this relationship between the structure types and the class types, see section 4.5.

One additional difficulty is that the inference algorithm requires that the types form a lattice, which most nominative type systems fail to do (this restriction allows the existence of a fast closure algorithm for typechecking; without it the inference problem is PSPACE-hard[10]). While seeming to form an additional difficulty, this restriction actually leads to a more elegant system and allows the system to be more expressive than most nominatively typed languages.

### 4.3.1 Class members

Classes also impose constraints on the types of their members. For instance, suppose a class Point is defined with two integer fields `x` and `y`, and the following function is written:

```
def get_x(pt:Point) do
  return pt.x
end
```

It should be possible to infer that the return type is `int`. This is done quite simply: when a class name appears in a typing constraint the system will not only add a constraint based on the object type constructor (section 4.5.3), but will also add a constraint for each field defined by the class.

When a subclass is derived from a class, it must be ensured that the subclass is in fact a subtype. This is performed by the polymorphic subsumption algorithm ($\leqslant^{\forall}$) described previously.

### 4.3.2 Constructors

Constructors for a class present an interesting problem. A subclass constructor must be able to call its parent class' constructor to initialise the parts of the object for which the parent is responsible. We must ensure that the constructor of the parent class is run before the child class performs any initialisation.

This is done in Java by syntactic restrictions: a call to `super()` must be the very first thing in the constructor's body. In Brick we adopt a slightly different approach: the call to the superclass's constructor is what actually returns the new object, and so it cannot be accessed before then by virtue of simply having no name (`self` is not in scope in a constructor).

### 4.3.3 Future work

It is intended that multiple conflicting definitions of a method (the standard problem of systems supporting multiple inheritance) be resolved using the C3 class linearisation algorithm first defined for the Dylan programming language; it is described fully in [3].

One other thing missing from the current model is a notion of data hiding: the members of an object are equally visible inside and outside the class's definition.

## 4.4 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. 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[4.2]. 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.

### 4.4.1 The value restriction

In languages like Haskell, every `let`-bound and all toplevel bindings are generalised[4.3]. This poses a well-known problem[35, ?] 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`!

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.

4.2. Although, by requiring type annotations in situations requiring first-class generalised bindings, the typing problem can be made tractable. See [30, 15] for examples.

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

There are a number of standard techniques used to mitigate this problem. Tofte's system[32], 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[19], 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*[35, 36] 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.

## 4.5 Integration of nominative and structural typing

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

Here is a concrete example of the problems that arise from this: 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
  x.draw()
end
```

```
def render(x: Shape) do
  x.draw()
end
```

```
def brandish(x: Cowboy) do
  x.draw()
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.

### 4.5.1 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.

### 4.5.2 Formal model of classes

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 2.5.1. 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.

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 $\boldsymbol{C}_1\,\textbf{extends}\,\boldsymbol{C}_2$ denoting the "directly extends" relation and $\textbf{defined}(\boldsymbol{C}_1)$ 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*[4.4]. $\textbf{superclasses}(\boldsymbol{C})$ 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.

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

4.4. Note that, under this definition, each class is a subclass of itself.

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.

### 4.5.3 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 2.5.1).

- 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[4.5].
- $\bot_{\mathbb{O}} \leqslant \boldsymbol{O}; \boldsymbol{O_1} \leqslant \boldsymbol{O_2} \Leftrightarrow \mathbf{classes}(\boldsymbol{O_1}) \supseteq \mathbf{classes}(\boldsymbol{O_2}) \wedge \mathbf{fields}(\boldsymbol{O_1}) \supseteq \mathbf{fields}(\boldsymbol{O_2})$
  $\bot_{\mathbb{O}}$ is a subtype of every type constructor.
  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 $\mathrm{arity}(\boldsymbol{O_1}) \supseteq \mathrm{arity}(\boldsymbol{O_2})$), 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 construction.

First, a few well-known lattice constructions:

- If $S$ is a set, then $S$ forms a lattice under $\subseteq$:
  $a \sqcap' b$ is $a \cap b$; $a \sqcup' b$ is $a \cup b$; $\bot'$ is $\varnothing$; $\top'$ is $S$
- If $S$ is partially ordered, then the upwards-closed subsets of $S$ form a lattice under $\subseteq$:

4.5. see section 4.1.1 for an extension of this providing for mutable variables

$a \sqcap' b$ is $a \cap b$; $a \sqcup' b$ is $a \cup b$; $\bot'$ is $\varnothing$; $\top'$ 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 $y \leqslant y'$: $(a, x) \sqcap' (b, y)$ is $(a \sqcap b, x \sqcap y)$; $(a, x) \sqcup' (b, y)$ is $(a \sqcup b, x \sqcup y)$; $\bot'$ is $(\bot, \bot)$; $\top'$ is $(\top, \top)$
- If $L$ is a lattice, the dual of $L$ (the same set with the ordering reversed) forms a lattice: $a \sqcap' b$ is $a \sqcup b$; $a \sqcup' b$ is $a \sqcap b$; $\bot'$ is $\top$; $\top'$ is $\bot$

**Lemma 4.1.** *If $L$ is a lattice, $s \in L$ and* $\text{erase}_s(L) = \{x | x \in L, x \not\leqslant s\} \cup \{\bot\}$*, 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 $\text{erase}_s$ that $x \not\leqslant s$ and $y \not\leqslant s$.

Let $a \in L = x \sqcup_L y$. Since $x \not\leqslant s$ and $y \not\leqslant s$, $a \not\leqslant s$ and so $a \in L'$. Thus, $\sqcup$ is well-defined on $L'$.

Let $b \in L = x \sqcap_L y$. If $b \not\leqslant s$, then $b \in L'$. Otherwise, since $b$ was the greatest lower bound, all lower bounds of $x$ and $y$ are $\leqslant s$. Thus, none of them (except $\bot$) appear in $L'$ and $\bot$ is the g.l.b. of $x$ and $y$ in $L'$. Thus, in either case, $\sqcap$ is well-defined on $L'$. □

$\text{erase}_s(L)$ 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: We take the space of upwards-closed sets of classes (which forms a lattice, see above), and the space of arbitrary sets of fields (which forms a lattice, see above). We take the space of pairs of these lattices (again, this forms a lattice), and we remove via $\text{erase}_s$ all of those points which include incompatible classes. Finally, we remove those points where the condition $\textbf{members}(\boldsymbol{C}) \subseteq \textbf{fields}(\boldsymbol{O})$. Since this condition preserves l.u.b. and g.l.b (it holds for $a$ and $b$ if and only if it holds for $a \sqcap b$ and $a \sqcup b$), the structure remains a lattice.

So, we have managed to reconstruct $\mathbb{O}$ in a way which is guaranteed to yield a lattice, hence $\mathbb{O}$ is a lattice.

### 4.5.4 Interface intersection types

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 interface 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[4.6].

4.6. 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.

# Part II:
# Language Implementation

*a kick in the monads*

# Chapter 5
# Implementation tools

*Let him choose out of my files, his projects to accomplish.*
***William Shakespeare,***
***"Coriolanus"***

## 5.1 Haskell

The pure, lazy, functional language Haskell was used for the implementation. There are two somewhat unusual features of Haskell which were used[5.1] heavily to describe the generic compiler infrastructure: laziness and monads.

### 5.1.1 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[5.2]. This tactic is used in the implementation of recursive functions, loops and similar recursive structures.

5.1. Abused

5.2. 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.

### 5.1.2 Monads

Monads are a central mechanism for expressing sequential code in Haskell. There are many, many, many introductions to the concept[34, 21, 14] 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.

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*. 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.

## 5.2 Happy and Alex

Happy[12] 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[6], 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.

## 5.3 LLVM

LLVM, or the Low-Level Virtual Machine, is "a compilation framework for lifelong program analysis and transformation"[18]. 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. However, none of these type-based optimisations have yet been implemented in Brick.

### 5.3.1 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 6
# Extending an interpreter

*All things are subject to interpretation ...*
***Friedrich Nietzsche***

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.

## 6.1 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.

For the purposes of this discussion, let's presume we represent abstract syntax trees for a very simple language with the following datatypes:

```
data Program  =  Seq Program Program
              |  Asgn Var Program
              |  If Exp Program
              |  While Exp Program
data Exp      =  EVar Var
              |  EAdd Exp Exp
              |  ECmpEq Exp Exp
```

Assume `Var` is a type representing variable names. The interpreter consists of two functions, `eval` and `evalExp`:

```
eval prog s1 = case prog of
    (Seq p1 p2) ->
      let s2 = eval p1 s1
          s3 = eval p2 s2
      in s3

    (Asgn v exp) ->
      let (s2,val) = evalExp exp s1
          s3 = varSet v val
      in s3

    (If exp p1 p2) ->
      let c = evalExpr exp s1
          condition = cond c
      in if condition then eval p1 s1
                      else eval p2 s1

    (While exp body) ->
      let c = evalExpr e s1
          condition = cond c
      in if condition then
           let s3 = eval body s2
           in eval prog s3
         else
           s2
```

```
evalExp exp s = case exp of
    (EVar v) -> varGet v s

    (EAdd e1 e2) ->
      let v1 = evalExp e1 s
          v2 = evalExp e2 s
      in opAdd v1 v2

    (ECmpEq e1 e2) ->
      let v1 = evalExp e1 s
          v2 = evalExp e2 s
      in opCmpEq v1 v2
```

`eval` evaluates a program, calling `evalExp` to evaluate an expression. The state (mapping of variables to values) is explicitly passed to `eval` and `evalExp`. Since `eval` may modify the state, `eval` is written to return a new state. `evalExp` never modifies the state, so returns a value instead[6.1].

A few primitives such as `varGet` and `varSet` (which read and modify a state) aren't shown here.

## 6.2 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. The monadic versions of `eval` and `evalExpr` look like:

6.1. Of course, this must change when side-effecting expressions are introduced. The language presented here is deliberately simplistic.

```
eval prog = case prog of
    (Seq p1 p2) -> do
      eval p1
      eval p2

    (Asgn v exp) -> do
      val <- evalExp exp
      varSet v val

    (If exp p1 p2) ->
      c <- evalExpr exp
      condition <- cond c
      in if condition then eval p1
                      else eval p2

    (While exp body) ->
      c <- evalExpr e
      condition <- cond c
      in if condition then do
           eval body
           eval prog
         else do
           return ()
```

```
evalExp exp = case exp of
    (EVar v) -> varGet v

    (EAdd e1 e2) -> do
      v1 <- evalExp e1
      v2 <- evalExp e2
      opAdd v1 v2

    (ECmpEq e1 e2) ->
      v1 <- evalExp e1
      v2 <- evalExp e2
      opCmpEq v1 v2
```

This is much clearer than the previous code. We do have to change the implementation of a few operations, though: `varGet` and `varSet` must now contain code to query and update the current monadic state.

As an added benefit, our `eval` function is now free from the implementation details of the state mapping. Since `eval` no longer passes around the state mapping directly, the interaction between `eval` and the state consists solely of calls to primitive operations. Thus, we can freely substitute other more efficient representations of this state without having to change the contents of `eval`.

and we can freely substitute other more efficient representations

##########

## 6.3 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.

## 6.4 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 7
# A compiler from an interpreter

*The soul is so far from being a monad that we have not only to interpret other souls to ourself but to interpret ourself to ourself.*

***T.S. Eliot***

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.

## 7.1 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.

## 7.2 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[7.1].

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.

### 7.2.1 Coalescing

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

7.1. 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.

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

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.

### 7.2.2 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.

### 7.2.3 Aside: Arrows

The original version of this work implemented the concepts not in terms of monads but in terms of arrows. Arrows are a generalisation of monads which can represent "computations with side-effects" as opposed to monads which represent "results with side effects", the difference being that an arrow can reason about a computation's inputs as well as its outputs[25, 13]. In this version of the work, the equivalent of `fixiter` arose more naturally.

In standard presentations of arrows, there are combinators to produce more complex arrows from simpler. Two important ones are:

$$\begin{aligned} \text{first} &:: \left(a \xrightarrow{\mathcal{A}} b\right) \mapsto \left((a,s) \xrightarrow{\mathcal{A}} (b,s)\right) \\ \text{left} &:: \left(a \xrightarrow{\mathcal{A}} b\right) \mapsto \left((a+s) \xrightarrow{\mathcal{A}} (b+s)\right) \end{aligned}$$

$a \xrightarrow{\mathcal{A}} b$ denotes a computation taking $a$ and producing $b$ in the arrow, $(a, s)$ is a pair of $a$ and $s$, while $a + s$ is a value which is either an $a$ or an $s$. The action of first is to produce an arrow which takes two inputs, passes the first through a given arrow and pass the second through unchanged. Left is analogous, except it passes only those values which match a condition through the given arrow and passes those which don't through unchanged.

First has a standard inverse, known as loop:

$$\text{loop} :: \left((a,s) \xrightarrow{\mathcal{A}} (b,s)\right) \mapsto \left(a \xrightarrow{\mathcal{A}} b\right)$$

This constructs an arrow whose second output is connected to its second input, and is used to model recursion in arrows. If we construct a similar inverse for left, we get:

$$\text{iter} :: \left((a+s) \xrightarrow{\mathcal{A}} (b+s)\right) \mapsto \left(a \xrightarrow{\mathcal{A}} b\right)$$

This constructs an arrow whose output is fed back to its input if it fails to meet a condition, and is a natural equivalent to `fixiter`.

## 7.3 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.[7.2] 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.

The trivial (and initially implemented) solution is simply to do a linear scan of the type table on each lookp, but more efficient solutions are possible. We propose a simple hashing scheme. This problem is related to the problems of storing sparse tables[31].

7.2. 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.

To avoid performing string comparisons, field names are hashed at compile time. To correctly handle hash collisions, each field is associated with a global variable with the same name. The system linker disambiguates globals, so we can rely on uniqueness and identify names with pointers to these globals variables.

The actual lookup scheme involves taking bitfields out of a precomputed hash value and using it to index into a table. Collisions are resolved by the global variable addresses. At the expense of some compilation time, we found it was generally possible to ensure (by searching the parameter space) that there are no more than 2 collisions for a given key, and so the lookup function could be implemented without branching.

Since LLVM is used to generate object code, we can rely on its optimisation passes to make changes like hoisting structure index operations out of loops.

## 7.4 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[7.3]. Both closures refer to the same box, and all of them alias the same copy of v.

## 7.5 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.

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

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 reuse 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 [4].

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.

$\mathcal{E}$ 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 primitive 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 $a \leqslant b \to c$. 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 (see appendix B for a full description of these typing rules).

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

## 7.6 Primitive operations

As a final description of the architecture of the implementation, this section contains a list of the operations available to `eval`. The various components simply implement these primtives, and the common language description in `eval` hooks them together.

Before describing the operations themselves, it is worth pointing out a few features conspicious by their absence:

**Control flow.** There are no intraprocedure control flow operations in the set of primitives, no if/then/else nor while nor return. Instead, `eval` implements such operations as though it were an interpreter (for instance, `break` and `continue` are implemented by `eval` throwing and catching exceptions), and the various implementations of the monad $\mathcal{M}$ contrive to ensure that the program states are correctly ordered.

**Bindings and name resolution.** The compiler, typechecker and interpreter core need not deal *at all* with variable names. The symbol table is entirely managed by the `eval` code. Primitives are provided for generating new variables (`varNew`), and it is up to `eval` to ensure these are passed around correctly.

**Closures.** Functions that access objects from outside their local scope are implemented transparently as Haskell closures. They are then "run" in the monad $\mathcal{M}$ to determine which items they access. The effect of this is that no special effort need be made to pass environment values to subfunctions in `eval`.

In the description of the operations to follow, $a \xrightarrow{\mathcal{M}} b$ is used to denote a function from $a$ to $b$ also performing a monadic action. It corresponds roughly to an action at the Brick language level, and the type in Haskell would be rendered `a -> M b`. Conversely, $a \mapsto b$ denotes a Haskell function, and corresponds roughly to an action taken during the compilation process.

As above, $\mathcal{E}$ denotes the (abstract) type of Brick values, while $\mathcal{V}$ denotes the abstract type of Brick variables. `Field` denotes a field name of a structure, it is simply a string. Some of these types do not correspond exactly to the Haskell types: these types are presented "uncurried" for simplicity.

| | | | |
|---|---|---|---|
| `litInt` | :: | $\texttt{Int} \xrightarrow{\mathcal{M}} \mathcal{E}$ | Integer literals |
| `litBool` | :: | $\texttt{Bool} \xrightarrow{\mathcal{M}} \mathcal{E}$ | Boolean literals |
| `voidValue` | :: | $() \xrightarrow{\mathcal{M}} \mathcal{E}$ | The unique literal value "`void`" |
| `cond` | :: | $\mathcal{E} \xrightarrow{\mathcal{M}} \texttt{Bool}$ | Evaluate a condition |
| `varNew` | :: | $() \xrightarrow{\mathcal{M}} \mathcal{V}$ | Create a new variable |
| `varGet` | :: | $\mathcal{V} \xrightarrow{\mathcal{M}} \mathcal{E}$ | Read a variable |
| `varSet` | :: | $(\mathcal{V}, \mathcal{E}) \xrightarrow{\mathcal{M}} ()$ | Write a variable |
| `structNew` | :: | $[\texttt{Field}] \xrightarrow{\mathcal{M}} \mathcal{E}$ | Create a new structure |
| `structGet` | :: | $(\mathcal{E}, \texttt{Field}) \xrightarrow{\mathcal{M}} \mathcal{E}$ | Read a structure field |
| `structSet` | :: | $(\mathcal{E}, \texttt{Field}) \xrightarrow{\mathcal{M}} \mathcal{E}$ | Write a structure field |
| `lambda` | :: | $\left(\mathcal{E} \xrightarrow{\mathcal{M}} \mathcal{E}\right) \xrightarrow{\mathcal{M}} \mathcal{E}$ | Create a new function |
| `apply` | :: | $\mathcal{E} \xrightarrow{\mathcal{M}} \left(\mathcal{E} \xrightarrow{\mathcal{M}} \mathcal{E}\right)$ | Apply a function |

For simplicity, some primitives have been omitted, including `letrec`, a list version of `lambda` which allows corecursive functions, `typeNew` and `typeConstrain` for checking user type annotations (ignored by the compiler and interpreter) and the primitives for describing and instantiating classes.

Any function which can be implemented in terms of the above primitives can be executed using the interpreter's definiton of those primitives, compiled using the compiler's, or typechecked using the typechecker's. Brick's parser, rather than produce a syntax tree, simply "executes" the program in an arbitrary monad $\mathcal{M}$ in terms of these primitives. The result of this "execution" can then be evaluated in any or all of the compiler components.

# Chapter 8
# Conclusions and future work

*I have seen the future and it doesn't work.*
***Robert Fulford***

We have shown that a practical imperative language can be constructed using the type inference engine described by Pottier and others, and that the problems that arise with the introduction of object-oriented features such as nominative classes and mutability are surmountable. We have also demonstrated some new techniques for a high-performance implementation of the type inference engine, and a novel means of combining the software-engineering and robustness advantages of nominative types with the quick-prototyping and flexibility advantages of structural types.

The implementation of Brick has shown that the development costs associated with a language implementation can be reduced by adopting a more abstract approach. In particular, complexities arising from symbol table lookup (or $\alpha$-renaming), control flow management, closure generation and others can be abstracted out into a single implementation which fulfills the needs of a compiler, interpreter, typechecker and possibly other as yet unimplemented phases.

## 8.1 Current state of the implementation

The interpreter and compiler can parse and run many programs, although some of the object-oriented features have not fully been implemented yet.

The type checker has been implemented more-or-less fully, and can infer and simplify types for some quite complex expressions (including recognising partial unrollings of a recursive type and correctly typing the Y combinator). Its performance (often a sore spot for complex inference algorithms) has so far been more than adequate, although it has yet to be tested on any large programs.

The "difficult problems" in the implementation of Brick are solved. However, in the quest to implement the more advanced parts of Brick's type system, compiler framework and runtime, some of the more mundane parts of a programming language were neglected. For instance, Brick does not yet support something as trivial as integer subtraction, and there is little support for input and output. These features are not complex to add, but time constraints during this project prevented their implementation.

## 8.2 Future work

Besides finishing the implementation of the features already described, other interesting areas of future work include:

**Error messages.** Currently Brick does not provide useful error messages on either syntax or type errors. This is an open research problem (particularly for complex type errors), but even basic support would be a boon to the current implementation.

**Garbage collection.** Writing a garbage collector was considered to be outside the scope of this project, and so compiled BRICK programs simply run until available memory is exhausted (interpreted programs are garbage-collected by Haskell's GC). This is obviously a vital addition before BRICK becomes a usable general-purpose language.

**Optimisation.** There is currently no mechanism by which the type inference engine can pass typing data to the compiler. Were this to be implemented, the compiler would have a large scope for type-based optimisations such as the elision of name-based field lookups.

**Foreign interface.** To be able to perform most I/O tasks, it is necessary that a language be able to integrate with the language of the host system (generally C). A means of easily generating a BRICK interface from a C interface would make the language much more useful.

# Appendix A
# BNF grammar for the syntax of Brick[A.1]

*This document describes the usage and input syntax of the Unix Vax-11 assembler* `as`. `as` *is designed for assembling code produced by the "C" compiler; certain concessions have been made to handle code written directly by people, but in general little sympathy has been extended*

***Berkeley Vax/Unix Assembler Reference Manual***

The grammar of Brick as used by the parser is deliberately more liberal than the syntax of valid Brick programs. The purpose of this is to lead to better error messages: although, say, an assignment statement is not valid in a class body, it is correctly parsed. Later during compilation, an error message will be generated stating that the assignment was misplaced. This means that misplaced statements can cause accurate error messages (rather than generic "= unexpected here" style messages) and the size of the grammar is reduced.

A number of current limitations of Brick are reflected in this grammar: for instance, functions can only take a single argument.

Program
  → Block
ClassDef
  → class ID TypeParamList
    SuperclassList TypeConstraintSet
    Stmts end
SuperclassList
  → $\epsilon$
  → '<:' InstantiatedClassList
InstantiatedClassList
  → InstantiatedClass
  → InstantiatedClassList ','
    InstantiatedClass
InstantiatedClass
  → ID TypeParamList
Exp
  → ID
  → Exp '.' Field
  → Int
  → '(' Exp ')'
  → ExpStmt
  → true
  → false
  → Struct
  → function '(' ArgBinder ')' FuncBody
ExpStmt
  → Exp '(' Exp ')'
Struct
  → '{' StructContents '}'
StructContents
  → $\epsilon$
  → DefList
DefElem
  → ID TypeAnn OptDef
OptDef
  → $\epsilon$
  → '=' Exp
DefList
  → DefElem
  → DefList ',' DefElem
Block
  → Stmts
Stmts
  → Stmt
  → Stmts ';' Stmt
Stmt
  → if Exp do Block IfTail
  → while Exp do Block end
  → do Block end
  → break
  → continue
  → return Exp

A.1. Title of this section may be inaccurate, see [5].

→ LValue '=' Exp
→ ExpStmt
→ var DefList
→ def DefList
→ def ID '(' ArgBinder ')' FuncBody

ArgBinder
→ ID

FuncBody
→ do Block end
→ '=' Exp

IfTail
→ end
→ elif Exp do Block IfTail
→ else do Block end

LValue
→ Var
→ Exp '.' Field

Var
→ ID

Field
→ ID

TypeAnn
→ $\epsilon$
→ ':' Type

Type
→ TypeTerm
→ TypeQuantifierSet

TypeTerm
→ TypeTerm '=>' TypeTerm
→ any
→ none
→ TypeVar
→ '(' TypeTerm ')'

TypeQuantifierSet
→ forall TypeVarList TypeConstraintSet

TypeConstraintSet
→ $\epsilon$
→ where TypeConstraintSet1

TypeConstraintSet1
→ TypeConstraint
→ TypeConstraintSet1 ',' TypeConstraint

TypeConstraint
→ TypeTerm '<:' TypeTerm

TypeParamList
→ $\epsilon$
→ '[' TypeVarList ']'

TypeVarList
→ TypeVar
→ TypeVarList ',' TypeVar

TypeVar
→ ID

# Appendix B
# Detailed typing rules for BRICK[B.1]

*A chic type, a rough type, an odd type - but never a stereotype*
***Jean-Michel Jarre***

We now present a formal specification of a subset of the typing rules of BRICK, to give a flavour of the internal workings of the constraint generator.

In the description of the typing rules, we will use a simplified, more functional-looking syntax. Programs will be represented as:

| | | | |
|---|---|---|---|
| e | = | $\lambda$x.e | $\lambda$-abstraction (ungeneralised binding) |
| | \| | e e | function application |
| | \| | let $\hat{\boldsymbol{x}}$ = e in e | let-abstraction (generalised binding) |
| | \| | {f1: e, f2: e, ...} | structure creation |
| | \| | e.f | structure field read |
| | \| | e.f := e | structure field write |
| | \| | e; e | sequential composition |

Bound names will be divided into two syntatically distinct classes: x, y, z, bound by $\lambda$, and $\hat{\boldsymbol{x}}$, $\hat{\boldsymbol{y}}$, $\hat{\boldsymbol{z}}$, bound by let (for a discussion of the distinction, including restrictions on what can be in the body of a let, see section 4.4). Mutable variables are omitted entirely, their effect is the same as that of a structure containing only one field.

Typing judgements will be of the form $\Gamma \vdash \text{e}: \tau \backslash C$, where e is the program being typed, $C$ is the resulting constraint graph, $\tau$ is the constructed type or variable within the constraint graph representing the type, and $\Gamma$ is the typing environment.

Typing environments $\Gamma$ will map names to types. For each ungeneralised variable x (bound by $\lambda$), $\Gamma(x)$ will be a single type variable $a$ which is constrained by each use of x. For each generalised variable $\hat{\boldsymbol{x}}$ (bound by let), $\Gamma(\hat{\boldsymbol{x}})$ will be an rc type $a \backslash C$, allowing the binding to be used polymorphically by duplicating constraints. The set of ungeneralised variables in $\Gamma$ is $\text{dom}_\lambda(\Gamma)$, and $\Gamma + (v \mapsto t)$ denotes the environment $\Gamma$ augmented with an extra mapping.

The subsumption relation on rc types is extended to include the typing environment as:

$$a \backslash C \leqslant^{\forall}_{\Gamma} a' \backslash C'$$

which is defined as:

$$a \backslash C \leqslant^{\forall} a' \backslash C' \quad \wedge \quad \Gamma(\text{x}) \backslash C \geqslant^{\forall} \Gamma(\text{x}) \backslash C' \text{ for all x} \in \text{dom}_\lambda(\Gamma)$$

That is, an rc type $A$ is a subtype of another rc type $B$ in an environment $\Gamma$ iff $A$ is a subtype of $B$ and each ungeneralised binding is assigned by $A$ a *supertype* of the type assigned by $B$. Since the ungeneralised bindings ($\lambda$-bound variables) are essentially extra inputs to the term, this means that a type is a subtype of another iff it gives a subtype to the current term while giving a *supertype* to the extra inputs. This is analogous to the contravariant typing rule for function inputs.

First, we present the *base typing rules* for BRICK:

B.1. Title of this section may be inaccurate, see [5].

$$\frac{\Gamma + (\mathrm{x} \mapsto a) \vdash \mathrm{e}: b \backslash C}{\Gamma \vdash \lambda \mathrm{x.e}: a \to b \backslash C} \quad \textsc{Lambda}$$

$$\frac{\Gamma \vdash \mathrm{e}_1: a \to b \backslash C \qquad \Gamma \vdash \mathrm{e}_2: a \backslash C}{\Gamma \vdash \mathrm{e}_1 \mathrm{e}_2: b \backslash C} \quad \textsc{Apply}$$

$$\frac{\Gamma \vdash \mathrm{e}_1: a_1 \backslash C \qquad \Gamma \vdash \mathrm{e}_2: a_2 \backslash C \qquad \ldots}{\Gamma \vdash \{\mathtt{f1}: \mathrm{e}_1, \mathtt{f2}: \mathrm{e}_2, \ldots\}: \{\mathtt{f1}: a_1/a_1, \mathtt{f2}: a_2/a_2, \ldots\} \backslash C} \quad \textsc{Struct-New}$$

$$\frac{\Gamma \vdash \mathrm{e}: \{\mathtt{f}: a/b\} \backslash C}{\Gamma \vdash \mathrm{e}.\mathtt{f}: b \backslash C} \quad \textsc{Struct-Get}$$

$$\frac{\Gamma \vdash \mathrm{e}_1: \{\mathtt{f}: a/b\} \backslash C \qquad \Gamma \vdash \mathrm{e}_2: a \backslash C}{\Gamma \vdash \mathrm{e}_1.\mathtt{f} := \mathrm{e}_2: () \backslash C} \quad \textsc{Struct-Set}$$

$$\frac{\Gamma \vdash \mathrm{e}_1: a \backslash C \qquad \Gamma \vdash \mathrm{e}_2: b \backslash C}{\Gamma \vdash \mathrm{e}_1; \mathrm{e}_2: b \backslash C} \quad \textsc{Seq}$$

$$\frac{\Gamma \vdash \mathrm{e}: a \backslash C \qquad a \backslash C \leqslant^{\forall}_{\Gamma} a' \backslash C'}{\Gamma \vdash \mathrm{e}: a' \backslash C'} \quad \textsc{Sub}$$

$$\frac{\Gamma(\mathrm{x}) = v}{\Gamma \vdash \mathrm{x}: v \backslash \varnothing} \quad \textsc{Var}$$

$$\frac{\Gamma \vdash \mathrm{e}_1: a \backslash C_1 \qquad \Gamma + (\hat{\boldsymbol{x}} \mapsto a \backslash C_1) \vdash \mathrm{e}_2: b \backslash C_2}{\Gamma \vdash \mathrm{let}\, \hat{\boldsymbol{x}} = \mathrm{e}_1 \,\mathrm{in}\, \mathrm{e}_2: b \backslash C_2} \quad \textsc{Let}$$

$$\frac{\Gamma(\hat{\boldsymbol{x}}) = a \backslash C}{\Gamma \vdash \hat{\boldsymbol{x}}: a \backslash C} \quad \textsc{Let-Var}$$

These typing rules, sadly, do not describe an algorithm. The rule SUB may be applied at any time, and since $C$ appears twice in the context for some rules it can't be easily calculated.

We now present the *inference rules* for BRICK. They are believed to be equivalent to the base typing rules, but no rigourous proof of this is provided.

In order to ensure that these new rules are syntax-directed, we need to eliminate all uses of the rule SUB. There are a couple of situations in which SUB is used: to allow subtypes to be substituted when typing a primitive operation (e.g. the argument to STRUCT-GET is allowed to have more than one field by virtue of SUB), to allow renaming of variables in generalised terms (SUB allows the constraints generated by a use of LET-VAR to be $\alpha$-renamed, provinding generalisation) and to allow simplfication of inferred type (any valid simplification can be justified by a use of SUB).

The first situation is avoided by removing all constructed terms from hypotheses of rules and building the minimum set of constraints explicitly. Since a minimal set of constraints is built, we end up with the typing derivation which corresponds to using SUB as little as possible to construct a valid derivation.

The second situation is resolved by handling renaming of generalised variables explicitly, whiile the third is resolved by introducing an explicit rule SIMPLIFY for performing simplifications which don't change the semantics of the graph. Its hypothesis demands that the two rc types be exactly equivalent ($\equiv^{\forall}_{\Gamma}$ is defined as the conjunction of $\leqslant^{\forall}_{\Gamma}$ and $\geqslant^{\forall}_{\Gamma}$), so its application is entirely optional: any valid derivation remains valid after all instances of SIMPLIFY are removed.

The type environment $\Gamma$ changes slightly in this set of rules: the types given to let-bound variables are no longer just rc types but rc types with a renaming: $\Gamma(\hat{\boldsymbol{x}}) = [\phi]a\backslash C$. The rc type $a\backslash C$ contains no variables in common with any other part of the constraint set, and the renaming $\phi$ shows the mapping from the ungeneralised variables ($\mathrm{dom}_\lambda(\Gamma)$) to the variables in $a\backslash C$. This allows the rc type $a\backslash C$ to apply constraints on the variables of $\mathrm{dom}_\lambda(\Gamma)$, without having them explicitly share variables as that would lead to very complex subtyping rules.

Renamings such as $\phi$, $\rho$ map variables to variables, and are generalised to rename constraint graphs and typing environments by renaming each of the variables within them.

$$\frac{\Gamma + (\mathrm{x} \mapsto a) \vdash^i \mathrm{e}: b\backslash C}{\Gamma \vdash \lambda \mathrm{x.e}: f\backslash C \oplus \{a \to b \leqslant f\}} \quad \textsc{Lambda}^i$$

$$\frac{\Gamma \vdash^i \mathrm{e}_1: f\backslash C_1 \qquad \Gamma \vdash^i \mathrm{e}_2: a\backslash C_2}{\Gamma \vdash^i \mathrm{e}_1\mathrm{e}_2: b\backslash C_1 \oplus \{f \leqslant a \to b\} \oplus C_2} \quad \textsc{Apply}^i$$

$$\frac{\Gamma \vdash^i \mathrm{e}_1: a_1\backslash C_1 \qquad \Gamma \vdash^i \mathrm{e}_2: a_2\backslash C_2 \qquad \ldots}{\Gamma \vdash^i \{\mathtt{f1}: \mathrm{e}_1, \mathtt{f2}: \mathrm{e}_2, \ldots\}: t\backslash\{\{\mathtt{f1}: a_1/a_1, \mathtt{f2}: a_2/a_2, \ldots\} \leqslant t\} \oplus \bigoplus_i C_i} \quad \textsc{Struct-New}^i$$

$$\frac{\Gamma \vdash^i \mathrm{e}: t\backslash C}{\Gamma \vdash^i \mathrm{e}.\mathtt{f}: b\backslash\{t \leqslant \{\mathtt{f}: a/b\}\} \oplus C} \quad \textsc{Struct-Get}^i$$

$$\frac{\Gamma \vdash^i \mathrm{e}_1: t\backslash C_1 \qquad \Gamma \vdash^i \mathrm{e}_2: a\backslash C_2}{\Gamma \vdash^i \mathrm{e}_1.\mathtt{f} := \mathrm{e}_2: c\backslash\{t \leqslant \{\mathtt{f}: a/b\}, () \leqslant c\} \oplus C_1 \oplus C_2} \quad \textsc{Struct-Set}^i$$

$$\frac{\Gamma \vdash^i \mathrm{e}_1: a\backslash C_1 \qquad \Gamma \vdash^i \mathrm{e}_2: b\backslash C_2}{\Gamma \vdash^i \mathrm{e}_1; \mathrm{e}_2: b\backslash C_1 \oplus C_2} \quad \textsc{Seq}^i$$

$$\frac{\Gamma \vdash \mathrm{e}: a\backslash C \qquad a\backslash C \equiv^\forall_\Gamma a'\backslash C'}{\Gamma \vdash \mathrm{e}: a'\backslash C'} \quad \textsc{Simplify}^i$$

$$\frac{\Gamma(\mathrm{x}) = v}{\Gamma \vdash^i \mathrm{x}: v\backslash\varnothing} \quad \textsc{Var}^i$$

$$\frac{\begin{array}{c}\phi \text{ a fresh renaming of } \mathrm{dom}_\lambda(\Gamma) \\ \phi(\Gamma) \vdash^i \mathrm{e}_1: a\backslash C_1 \qquad \Gamma + (\hat{\boldsymbol{x}} \mapsto [\phi]a\backslash C_1) \vdash \mathrm{e}_2: b\backslash C_2\end{array}}{\Gamma \vdash \mathrm{let}\ \hat{\boldsymbol{x}} = \mathrm{e}_1\ \mathrm{in}\ \mathrm{e}_2: b\backslash C_2} \quad \textsc{Let}^i$$

$$\frac{\Gamma(\hat{\boldsymbol{x}}) = [\phi]a\backslash C \qquad \rho \text{ a fresh renaming of } \mathrm{fv}(a\backslash C)}{\Gamma \vdash \hat{\boldsymbol{x}}: \rho(a)\backslash\rho(C) \oplus \{v \leqslant \rho(\phi(v)) | v \in \mathrm{dom}(\phi)\}} \quad \textsc{Let-Var}^i$$

Note also that these typing rules correspond to the primitives defined in section 7.6. Indeed, the typechecker is an implementation of that chapter's monad $\mathcal{M}$ where the action of the monad is to collect a constraint graph and each primitive is implemented as one of the above typing rules and simply adds more constraints to the graph.

# 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]** K. Barrett, B. Cassels, P. Haahr, D.A. Moon, K. Playford, and P.T. Withington. A monotonic superclass linearization for Dylan. pages 69–82, 1996.

**[4]** P. Cousot. Types as abstract interpretations. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 316–331. ACM, 1997.

**[5]** R. Dolan, C. Dolan, and K. Dolan. Get well soon. Card, 2011. after appendectomy.

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

**[7]** G. Dubochet and M. Odersky. Compiling structural types on the JVM: a comparison of reflective and generative techniques from Scala's perspective. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 34–41. ACM, 2009.

**[8]** J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. *ACM SIGPLAN Notices*, 30(10):169–184, 1995.

**[9]** 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.

**[10]** Alexandre Frey. Satisfying subtype inequalities in polynomial space. *Static Analysis*, 1302:265–277, 1997. 10.1007/BFb0032747.

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

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

**[13]** J. Hughes. Generalising monads to arrows. *Science of computer programming*, 37(1-3):67–111, 2000.

**[14]** S.L.P. Jones and P. Wadler. Imperative functional programming. 1993.

**[15]** S.P. Jones, D. Vytiniotis, S. Weirich, and M. Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17(01):1–82, 2007.

**[16]** Stefan Kaes. Type inference in the presence of overloading, subtyping and recursive types. In *LFP '92: Proceedings of the 1992 ACM conference on LISP and functional programming*, pages 193–204, New York, NY, USA, 1992. ACM.

**[17]** Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient recursive subtyping. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 419–428, New York, NY, USA, 1993. ACM.

**[18]** Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

**[19]** 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.

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

**[21]** E. Moggi and University of Edinburgh. Laboratory for Foundation of Computer Science. *An abstract view of programming languages*. University of Edinburgh, Laboratory for Foundation of Computer Science, 1990.

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

**[23]** J. Palsberg and S. Smith. Constrained types and their expressiveness. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(5):519–527, 1996.

**[24]** Jens Palsberg and Patrick O'Keefe. A type system equivalent to flow analysis. *ACM Trans. Program. Lang. Syst.*, 17(4):576–599, 1995.

**[25]** R. Paterson. Arrows and computation. *The Fun of Programming*, pages 201–222, 2003.

**[26]** S. Peyton Jones, S. Marlow, and C. Elliott. Stretching the storage manager: weak pointers and stable names in Haskell. *Implementation of Functional Languages*, pages 37–58, 2000.

**[27]** F. Pottier. Simplifying subtyping constraints. In *Proceedings of the first ACM SIGPLAN international conference on Functional programming*, pages 122–133. ACM, 1996.

**[28]** François Pottier. Type inference in the presence of subtyping: from theory to practice. PhD Thesis, INRIA, 1998.

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

**[30]** C.V. Russo and D. Vytiniotis. QML: explicit first-class polymorphism for ML. In *Proceedings of the 2009 ACM SIGPLAN workshop on ML*, pages 3–14. ACM, 2009.

**[31]** R.E. Tarjan and A.C.C. Yao. Storing a sparse table. *Communications of the ACM*, 22(11):606–611, 1979.

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

**[33]** Valery Trifonov and Scott Smith. Subtyping constrained types. *Static Analysis*, 1145:349–365, 1996.

**[34]** P. Wadler. Monads for functional programming. *Advanced Functional Programming*, pages 24–52, 1995.

**[35]** A.K. Wright. Polymorphism for imperative languages without imperative types. Technical report, Rice University Dept. of Computer Science, 1993.

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