Principles of Programming Languages
Version 1.0.2

Mike Grant
Zachary Palmer
Scott Smith

http://pl.cs.jhu.edu/pl/book

This document was last compiled on February 10, 2018.

Contents

Preface

This book is an introduction to the study of programming languages. The material has evolved from lecture notes used in a programming languages course for juniors, seniors, and graduate students at Johns Hopkins University [21].

The book treats programming language topics from a foundational. It is foundational in that it focuses on core concepts in language design such as functions, records, objects, and types and not directly on applied languages such as C, C++, or Java. We show how the particular core concepts are realized in these modern languages, and so the reader should emerge from this book with a stronger sense of how they are structured.

While the book uses formal mathematical techniques such as operational semantics and type systems, it does not emphasize proofs of properties of these systems. We will sketch the intuitions of some properties but not do any detailed proofs.

The OCaml Language

The OCaml programming language [15] is used throughout the book, and assignments related to the book are best written in OCaml. OCaml is a modern dialect of ML which has the advantages of being reliable, fast, free, and available on just about any platform through `http://caml.inria.fr`.

The FbDK

Complementing the book is the **F♭** Development Kit, FbDK. It is a set of OCaml utilities and interpreters for designing and experimenting with the toy **F♭** and **F♭SR** languages defined in the book. It is available from the book homepage at `http://pl.cs.jhu.edu/pl/book`, and is documented in Appendix **??**.

Background Needed

The book assumes familiarity with the basics of OCaml, including the module system (but not the objects, the "O" in OCaml). Beyond that there is no absolute prerequisite, but knowledge of C, C++, and Java is helpful because many of the topics in this book are implemented in these languages. The compiler presented in chapter 8 produces C code as its target, and so a basic knowledge of C will be needed to implement the compiler. More nebulously, a certain "mathematical maturity" greatly helps in understanding the concepts, some of which are deep. for this reason, previous study of mathematics, formal logic and other foundational topics in Computer Science such as automata theory, grammars, and algorithms will be a great help.

Chapter 1

Introduction

General-purpose computers have the amazing property that a single piece of hardware can do any computation imaginable. Before general-purpose computers existed, there were special-purpose computers for arithmetic calculations, which had to be manually reconfigured to carry out different calculations. A general-purpose computer, on the other hand, has the configuration information for the calculation in the computer memory itself, in the form of a *program*. The designers realized that if they equipped the computer with the program instructions to access an arbitrary memory location, instructions to branch to a different part of the program based on a condition, and the ability to perform basic arithmetic, then any computation they desired to perform was possible within the limits of how much memory, and patience waiting for the result, they had.

These initial computer programs were in machine language, a sequence of bit patterns. Humans understood this language as assembly language, a textual version of the bit patterns. So, these machine languages were the first programming languages, and went hand-in-hand with general-purpose computers. So, programming languages are a fundamental aspect of general-purpose computing, in contrast with *e.g.*, networks, operating systems, and databases.

1.1 The Pre-History of Programming Languages

The concept of general-purpose programming in fact predates the development of computers. In the field of mathematical logic in the early 20th century, logicians created their own programming languages. Their motivation originally sprang from the concept of a *proof system*, a set of rules in which logical truths could be derived, mechanically. Since proof rules can be applied mechanically, all of the logically true facts can be mechanically enumerated by a person sitting there applying all of the rules in every order possible. This means the set of provable truths are *recursively enumerable*. Logicians including Frege, Church, and Curry wanted to create a more general theory of logic and proof; this led Church to define the λ-calculus in 1932, an abstract language of functions which also defined a logic. The logic turned out to be inconsistent, but by then logicians had discovered that the idea of a theory of functions and their (abstract) computations was itself of interest. They found that some interesting logical properties (such as the collection of all truths in certain logical systems) were in fact not recursively enumerable,

meaning no computer program could ever enumerate them all. So, the notion of general-purpose computation was first explored in the abstract by logicians, and only later by computer designers. The λ-calculus is in fact a general-purpose programming language, and the concept of higher-order functions, introduced in the Lisp programming language in the 1960's, was derived from the higher-order functions found in the λ-calculus.

1.2 A Brief Early History of Languages

There is a rich history of programming languages that is well worth reading about; here we provide a terse overview.

The original computer programming languages, as mentioned above, were so-called machine languages: the human and computer programmed in same language. Machine language is great for computers but not so great for humans since the instructions are each very simple and so many, many instructions are required. High-level languages were introduced for ease of programmability by humans. FORTRAN was the first high-level language, developed in 1957 by a team led by Backus at IBM. FORTRAN programs were translated (*compiled*) into machine language to be executed. They didn't run as fast as hand-coded machine language programs, but FORTRAN nonetheless caught on very quickly because FORTRAN programmers were much more productive. A swarm of early languages followed: ALGOL in '58, Lisp in the early 60's, PL/1 in the late 60's, and BASIC in 1966.

Languages have developed on many fronts, but there is arguably a major thread of evolution of languages in the following tiers:

1. Machine language: program directly in the language of the computer
2. FORTRAN, BASIC, C, Pascal, ...: first-order functions, nested control structures, arrays.
3. Lisp, Scheme, ML: higher-order functions, automated garbage collection, memory safety; strong typing in ML

Object-oriented language development paralleled this hierarchy.

1. (There was never an object-oriented machine language)
2. Simula67 was the original object-oriented language, created for simulation. It was FORTAN-like otherwise. C++ is another first-order object-oriented language.
3. Smalltalk in the late 70's: Smalltalk is a higher-order object-oriented language which also greatly advanced the concept of object-oriented programming by showing its applicability to GUI programming. Java is partly higher order, has automated garbage collection, and is strongly typed.

Domain-specific programming languages (DSLs) are languages designed to solve a more narrow domain of problems. All languages are at least domain-*specialized:* FORTRAN is most highly suited to scientific programming, Smalltalk for GUI programming, Java for Internet programming, C for UNIX system programming, Visual Basic for Microsoft Windows. Some languages are particularly narrow in applicability; these are

called *Domain-specific languages*. SNOBOL and Perl are text processing languages. UNIX shells such as sh and csh are for simple scripting and file and text hacking. Prolog is useful for implementing rule-based systems. ML is to some degree a DSL for language processing. Also, some languages aren't designed for general programming at all, in that they don't support full programmability via iteration and arbitrary storage allocation. SQL is a database query language; XML is a data representation language.

1.3 This Book

In this book, our goal is to study the fundamental concepts in programming languages, as opposed to learning a wide range of languages. Languages are easy to learn, it is the concepts behind them that are difficult. The basic features we study in turn include higher-order functions, data structures in the form of records and variants, mutable state, exceptions, objects and classes, and types. We also study language implementations, both through language interpreters and language compilers. Throughout the book we write small interpreters for toy languages, and in Chapter 8 we write a principled compiler. We define type checkers to define which programs are well-typed and which are not. We also take a more precise, mathematical view of interpreters and type checkers, via the concepts of *operational semantics* and *type systems*. These last two concepts have historically evolved from the logician's view of programming.

Now, make sure your seat belts are buckled, sit back, relax, and enjoy the ride...

Chapter 2

Operational Semantics

2.1 A First Look at Operational Semantics

The **syntax** of a programming language is the set of rules governing the formation of expressions in the language. The **semantics** of a programming language is the *meaning* of those expressions.

There are several forms of language semantics. Axiomatic semantics is a set of axiomatic truths in a programming language. Denotational semantics involves modeling programs as static mathematical objects, namely as set-theoretic functions with specific properties. We, however, will focus on a form of semantics called operational semantics.

An operational semantics is a mathematical model of programming language *execution*. It is, in essence, an interpreter defined mathematically. However, an operational semantics is more precise than an interpreter because it is defined mathematically, and not based on the meaning of the programming language in which the interpreter is written. This might sound sound like a pedantic distinction, but interpreters interpret e.g. a language's `if` statements with the `if` statement of the language the interpreter is written in. This is in some sense a circular definition of `if`. Formally, we can define operational semantics as follows.

Definition 2.1 (Operational Semantics)**.** *An* ***operational semantics*** *for a programming language is a mathematical definition of its computation relation, $e \Rightarrow v$, where e is a program in the language.*

$e \Rightarrow v$ is mathematically a 2-place relation between expressions of the language, e, and values of the language, v. Integers and booleans are values. Functions are also values because they don't compute to anything. e and v are **metavariables**, meaning they denote an arbitrary expression or value, and should not be confused with the (regular) variables that are part of programs.

An operational semantics for a programming language is a means for understanding in precise detail the meaning of an expression in the language. It is the formal specification of the language that is used when writing compilers and interpreters, and it allows us to rigorously verify things about the language.

2.2 BNF grammars and Syntax

Before getting into meaning we need to take a step back and first precisely define language syntax. This is done with formal grammars. *Backus-Naur Form* (BNF) is a standard grammar formalism for defining language syntax. You could well be familiar with BNF since it is often taught in introductory courses, but if not we provide a brief overview. All BNF grammars comprise *terminals*, *nonterminals (aka syntactic categories)*, and production rules. Terminals are traditionally identified using lower-case letters; non-terminals are identified using upper-case letters. Production rules describe how non-terminals are defined. The general form of production rules is:

$$\langle \text{nonterminal} \rangle ::= \langle \text{form 1} \rangle \mid \cdots \mid \langle \text{form n} \rangle$$

where each "form" above describes a particular language form – that is, a string of terminals and non-terminals. A *term* in the language is a string of terminals which matches the description of one of these rules (traditionally the first).

For example, consider the language Sheep. Let $\{S\}$ be the set of nonterminals, $\{a, b\}$ be the set of terminals, and the grammar definition be:

$$S ::= b \mid Sa$$

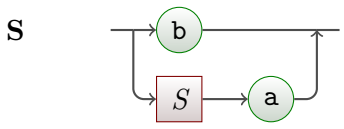
Note that this is a recursive definition. Examples of terms in Sheep are

$$b, ba, baa, baaa, baaaa, \ldots$$

That is, any string starting with the character b and followed by zero or more a characters is a term in Sheep. The following are examples that are not terms in SHEEP:

- a: Terms in Sheep must start with a b.
- $bbaaa$: Sheep does not allow multiple b characters in a term.
- $baah$: h is not a terminal in Sheep.
- $Saaa$: S is a non-terminal in Sheep. Terms may not contain non-terminals.

Another way of expressing a grammar is by the use of a **syntax diagram**. Syntax diagrams describe the grammar visually rather than in a textual form. For example, the following is a syntax diagram for the language Sheep:



The above syntax diagram describes all terms of the Sheep language. To generate a form of S, one starts at the left side of the diagram and moves until one reaches the right. The rectangular nodes represent non-terminals while the rounded nodes represent terminals. Upon reaching a non-terminal node, one must construct a term using that non-terminal to proceed.

As another example, consider the language Frog. Let $\{F, G\}$ be the set of nonterminals, $\{r, i, b, t\}$ be the set of terminals, and the grammar definition be:

$$F ::= rF \mid iG$$
$$G ::= bG \mid bF \mid t$$

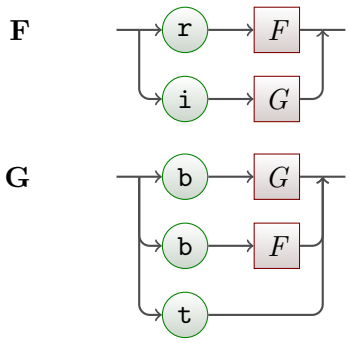
Note that this is a mutually recursive definition. Note also that each production rule defines a syntactic category. Terms in FROG include:

$$ibit, ribbit, ribibibbbit \ldots$$

The following terms are not terms in Frog:

- *rbt*: When a term in Frog starts with r, the following non-terminal is F. The non-terminal F may only be exapnded into rF or iG, neither of which start with b. Thus, no string starting with rb is a term in Frog.
- *rabbit*: a is not a terminal in Frog.
- *rrrrrrF*: F is a non-terminal in Frog; terms may not contain non-terminals.
- *bit*: The only forms starting with b appear as part of the definition of G. As F is the first non-terminal defined, terms in Frog must match F (which does not have any forms starting with b).

The following syntax diagram describes Frog:



2.2.1 Operational Semantics for Logic Expressions

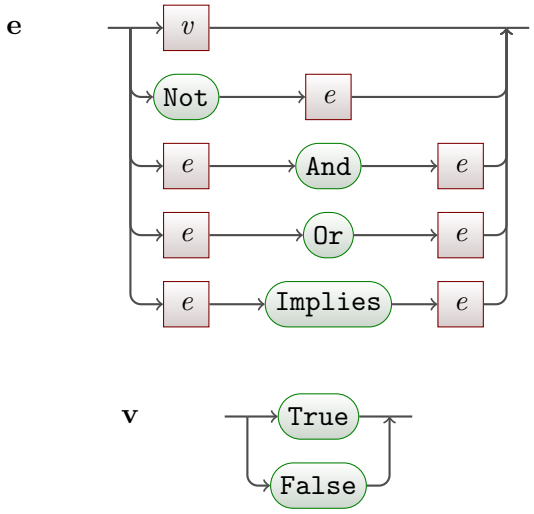
In order to get a feel for what an operational semantics is and how it is defined, we will now examine the operational semantics for a very simple language: propositional boolean logic with no variables. The syntax of this language is as follows. An expression e is recursively defined to consist of the values `True` and `False`, and the expressions e `And` e, e `Or` e, e `Implies` e, and `Not` e.[1] This syntax is known as the **concrete syntax**,

[1] Throughout the book we use syntax very similar to OCaml in our toy languages, but with the convention of capitalizing keywords to avoid potential conflicts with the OCaml language.

because it is the syntax that describes the textual representation of an expression in the language. We can express it in a BNF grammar as follows:

$$\begin{array}{lcll} e & ::= & v \mid \texttt{Not}\ e \mid e\ \texttt{And}\ e \mid e\ \texttt{Or}\ e \mid e\ \texttt{Implies}\ e \mid (e) & \textit{expressions} \\ v & ::= & \texttt{True} \mid \texttt{False} & \textit{values} \end{array}$$

The following is an equivalent syntax diagram:



Note that the syntax above breaks tradition somewhat by using lower-case letters for non-terminals. Terminals are printed in fixed-width font. The rationale for this is consistency with the metavariables we will be using in operational semantics below and will become clear shortly.

We can now discuss the operational semantics of the boolean language. Operational semantics are written in the form of logic rules, which are written as a series of pre-conditions above a horizontal line and the conclusion below it. For example, the logic rule

$$\textit{(Apple Rule)} \qquad \frac{\text{Red}(x) \quad \text{Shiny}(x)}{\text{Apple}(x)}$$

indicates that if a thing is red and shiny, then that thing is an apple. This is, of course, not true; many red, shiny things exist which are not apples. Nonetheless, it is a valid logical statement. In our work, we will be defining logical rules pertaining to a programming language; as a result, we have control over the space in which the rules are constructed. We need not necessarily concern ourselves with intuitive sense so long as the programming language has a mathematical foundation.

Operational semantics rules discuss how pieces of code evaluate. For example, let us consider the `And` rule. We may define the following rule for `And`:

$$\textit{(And Rule (Try 1))} \qquad \frac{}{\texttt{True And False} \Rightarrow \texttt{False}}$$

This rule indicates that the boolean language code `True And False` evaluates to `False`. The absence of any preconditions above the line means that no conditions must be met; this operational semantics rule is always true. Rules with nothing above the line are termed **axioms** since they have no preconditions and so the conclusion always holds.

As a rule, though, it isn't very useful. It only evaluates a very specific program. This rule does not describe how to evaluate the program `True And True`, for instance. In order to generalize our rules to describe a full language and not just specific terms within the language, we must make use of metavariables.

To maintain consistency with the above BNF grammar, we use metavariables starting with e to represent expressions and metavariables starting with v to represent values. We are now ready to make an attempt at describing every aspect of the `And` operator using the following new rule:

(And Rule (Try 2)) $$\frac{}{v_1 \texttt{ And } v_2 \Rightarrow \text{the logical and of } v_1 \text{ and } v_2}$$

Using this rule, we can successfully evaluate `True And False`, `True and True`, and so on. Note that we have used a textual description to indicate the value of the expression v_1 `And` v_2; this is permitted, although most rules in more complex languages will not use such descriptions.

We very quickly encounter limitations in our approach, however. Consider the program `True And (False And True)`. If we tried to apply the above rule to that program, we would have $v_1 =$ `True` and $v_2 =$ `(False And True)`. These two values cannot be applied to logical and as `(False and True)` is not a boolean value; it is an expression. Our boolean language rule does not allow for cases in which the operands to `And` are expressions. We therefore make another attempt at the rule:

(And Rule (Try 3)) $$\frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2}{e_1 \texttt{ And } e_2 \Rightarrow \text{the logical and of } v_1 \text{ and } v_2}$$

This rule is almost precisely what we want; in fact, the rule itself is complete. Intuitively, this rule says that e_1 `And` e_2 evaluates to the logical and of the values represented by e_1 and e_2. But consider again the program `True And False`, which we expect to evaluate to `False`. We can see that $e_1 =$ `True` and that $e_2 =$ `False`, but our evaluation relation does not relate v_1 or v_2 to any value. This is because, strictly speaking, we do not know that `True` $\Rightarrow$ `True`.

Of course, we would like that to be the case and, since we are in the process of defining the language, we can make it so. We simply need to declare it in an operational semantics rule.

(Value Rule) $$\frac{}{v \Rightarrow v}$$

The value rule above is an axiom declaring that any value always evaluates to itself. This satisfies our requirement and allows us to make use of the `And` rule. Using this formal logic approach, we can now prove that `True And (False And True)` $\Rightarrow$ `False` as follows:

$$\dfrac{\dfrac{}{\texttt{True} \Rightarrow \texttt{True}} \quad \dfrac{\dfrac{}{\texttt{False} \Rightarrow \texttt{False}} \quad \dfrac{}{\texttt{True} \Rightarrow \texttt{True}}}{\texttt{False And True} \Rightarrow \texttt{False}}}{\texttt{True And (False And True)} \Rightarrow \texttt{False}}$$

One may read the above **proof tree** as an explanation as to why `True And (False And True)` evaluates to `False`. We can choose to read that proof as follows: *"`True And (False And True)` evaluates to `False` by the `And` rule because we know `True` evaluates to `True`, that `False And True` evaluates to `False`, and that the logical and of true and false is false. We know that `False And True` evaluates to `False` by the `And` rule because `True` evaluates to `True`, `False` evaluates to `False`, and the logical and of true and false is false."*

An equivalent and similarly informal format for the above is:

`True And (False And True)` $\Rightarrow$ `False`, because by the `And` rule
  `True` $\Rightarrow$ `True`, and
  `(False And True)` $\Rightarrow$ `False`, the latter because
    `True` $\Rightarrow$ `True`, and
    `False` $\Rightarrow$ `False`

The important thing to note about all three of these representations is that they are describing a proof tree. The proof tree consists of nodes which represent the application of logical rules with preconditions as their children. To complete our boolean language, we define the $\Rightarrow$ relation using a complete set of operational semantics rules:

(Value Rule) $\dfrac{}{v \Rightarrow v}$

(Not Rule) $\dfrac{e \Rightarrow v}{\texttt{Not } e \Rightarrow \text{the negation of } v}$

(And Rule) $\dfrac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2}{e_1 \texttt{ And } e_2 \Rightarrow \text{the logical and of } v_1 \text{ and } v_2}$

The rules for `Or` and `Implies` are left as an exercise to the reader (see Exercise 2.4).

These rules form a **proof system** as is found in mathematical logic. Logical rules express incontrovertible logical truths. A **proof** of $e \Rightarrow v$ amounts to constructing a sequence of rule applications such that, for any given application of a rule, the items above the line appeared earlier in the sequence and such that the final rule application is $e \Rightarrow v$. A proof is structurally a tree, where each node is a rule, and the subtree rules have conclusions which exactly match what the parent's assumptions are. For a proof

tree of $e \Rightarrow v$, the root rule has as its conclusion $e \Rightarrow v$. Note that *all leaves of a proof tree must be axioms*. A tree with a non-axiom leaf is not a proof.

Notice how the above proof tree is expressing how this logic expression could be computed. Proofs of $e \Rightarrow v$ corresponds closely to how the execution of e produces the value v as result. The only difference is that "execution" starts with e and produces the v, whereas a proof tree describes a relation between e and v, not a function from e to v.

Lemma 2.1. *The boolean language is **deterministic**: if $e \Rightarrow v$ and $e \Rightarrow v'$, then $v = v'$.*

Proof. By induction on the height of the proof tree. □

Lemma 2.2. *The boolean language is **normalizing**: For all boolean expressions e, there is some value v where $e \Rightarrow v$.*

Proof. By induction on the size of e. □

When a proof $e \Rightarrow v$ can be constructed for some program e, we say that e **converges**. When no such proof exists, e **diverges**. Because the boolean language is normalizing, all programs in that language are said to converge. Some languages (such as OCaml) are not normalizing; there are syntactically legal programs for which no evaluation proof exists. An example of a OCaml program which is divergent is `let rec f x = f x in f 0;;`.

2.2.2 Abstract Syntax

Our operational semantics rules have expressed the evaluation relation in terms of concrete syntax using metavariables. Operators, such as the infix operator `And`, have appeared in textual format. This is a good representation for humans to read because it appeals to our intuition; it is not, however, an ideal computational representation. We read `True And False` as "perform a logical and with operands `True` and `False`". We read `True And (False And True)` as "perform a logical and with operands `False` and `True` and then perform a logical and with operands `True` and the result of the last operation." If we are to write programs (such as interpreters) to work with our language, we need a representation which more accurately describes how we think about the program.

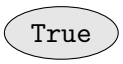
The **abstract syntax** of a language is such a representation. A term in an abstract syntax is represented as a **syntax tree** in which each operation to be performed is a node and each operand to that operation is a child of that node. In order to represent abstract syntax trees for the boolean language, we might use the following OCaml data type:

```
type boolexp =
    True | False |
    Not of boolexp |
    And of boolexp * boolexp |
    Or of boolexp * boolexp |
    Implies of boolexp * boolexp;;
```

To understand how the abstract and concrete syntax relate, consider the following examples:

Example 2.1.

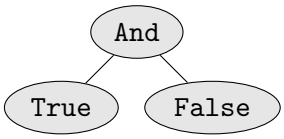
Concrete:
`True`



Abstract:
`True`

Example 2.2.

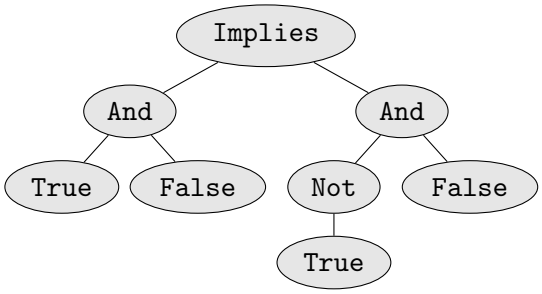
Concrete:
`True And False`



Abstract:
`And(True, False)`

Example 2.3.

Concrete:
`(True And False) Implies`
`((Not True) And False)`



Abstract:
`Implies( And(True,False) ,`
`And(Not(True),False) )`

There is a simple and direct relationship between the concrete syntax of a language and the abstract syntax. As mentioned above, the abstract syntax is a form which more directly represents the operations being performed whereas the concrete syntax is the form in which the operations are actually expressed. Part of the process of compiling or interpreting a program is to translate the concrete syntax term (source file) into an abstract syntax term (AST) in order to manipulate it. We define a relation $[\![c]\!] = a$ to map concrete syntax form c to abstract syntax form a (in this case for the boolean language):

$$
\begin{array}{rcl}
[\![\texttt{True}]\!] & = & \texttt{True} \\
[\![\texttt{False}]\!] & = & \texttt{False} \\
[\![\texttt{Not } e]\!] & = & \texttt{Not}(e) \\
[\![e_1 \texttt{ And } e_2]\!] & = & \texttt{And}([\![e_1]\!], [\![e_2]\!]) \\
[\![e_1 \texttt{ Or } e_2]\!] & = & \texttt{Or}([\![e_1]\!], [\![e_2]\!]) \\
[\![e_1 \texttt{ Implies } e_2]\!] & = & \texttt{Implies}([\![e_1]\!], [\![e_2]\!])
\end{array}
$$

For example, this relation indicates the following:

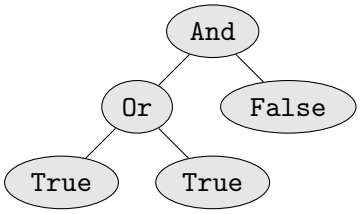
```
⟦(True And False) Implies ((Not True) And False)⟧
  =  Implies( ⟦True And False⟧, ⟦(Not True) And False⟧ )
  =  Implies( And(⟦True⟧, ⟦False⟧), And(⟦Not True⟧, ⟦False⟧) )
  =  Implies( And(True, False), And(Not(⟦True⟧), False) )
  =  Implies( And(True, False), And(Not(True), False) )
```

The grammar we give is ambiguous in that there are multiple parse trees for some concrete expressions, but we implicitly assume the usual operator precedence applies with And binding tighter than Or binding tighter than Implies. Consider the following examples:

Example 2.4.

Concrete:
`True Or True And False`

Abstract:
`And(Or(True,True),False)`

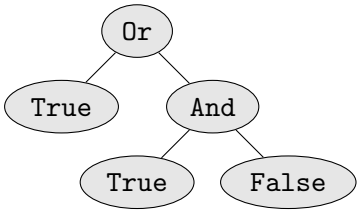


$$\dfrac{\dfrac{}{\texttt{True} \Rightarrow \texttt{True}} \quad \dfrac{}{\texttt{True} \Rightarrow \texttt{True}}}{\texttt{True Or True} \Rightarrow \texttt{True}} \quad \dfrac{}{\texttt{False} \Rightarrow \texttt{False}}}{\texttt{True Or True And False} \Rightarrow \texttt{False}}$$

Example 2.5.

Concrete:
`True Or (True And False)`

Abstract:
`Or(True,And(True,False))`



$$\dfrac{\dfrac{}{\texttt{True} \Rightarrow \texttt{True}} \quad \dfrac{\dfrac{}{\texttt{True} \Rightarrow \texttt{True}} \quad \dfrac{}{\texttt{False} \Rightarrow \texttt{False}}}{\texttt{True And False} \Rightarrow \texttt{False}}}{\texttt{True Or (True And False)} \Rightarrow \texttt{True}}$$

The expression in example 2.4 will evaluate to `False` because one must evaluate the `Or` operation first and then evaluate the `And` operation using the result. Example 2.5, on the other hand, performs the operations in the opposite order. Note that in both examples, though, the parentheses themselves are no longer overtly present in the abstract syntax. This is because they are implicitly represented in the structure of the AST; that is, the AST in example 2.5 would not have the shape that it has if the parentheses were not present in the concrete syntax of the form.

In short, parentheses merely change how expressions are grouped. In example 2.5, the only rule we can match to the *entire expression* is the `Or` rule; the `And` rule obviously can't match because the left parentheses would be part of e_1 while the right parenthesis would be part of e_2 (and expressions with unmatched parentheses make no sense). Similarly but less obviously, example 2.4 can only match the `And` rule; the associativity implicitly forces the `Or` rule to happen first, giving the `And` operator that entire expression to evaluate. This distinction is clearly and correspondingly represented in the ASTs of the examples, a fact which is key to the applicability of operational semantics.

2.2.3 Operational Semantics and Interpreters

As alluded above, there is a very close relationship between an operational semantics and an actual interpreter written in OCaml. Given an operational semantics defined via the relation $\Rightarrow$, there is a corresponding (OCaml) evaluator function `eval`.

Definition 2.2 (Faithful Implementation)**.** *A (OCaml) interpreter function eval faithfully implements an operational semantics $e \Rightarrow v$ if:*
$e \Rightarrow v$ if and only if `eval`$(\llbracket e \rrbracket)$ *returns result* $\llbracket v \rrbracket$.

To demonstrate this relationship, we will demonstrate the creation of an `eval` function in OCaml. Our first draft of the function will, for sake of simplicity, only consist of the `And` rule and the value rule:

```
let eval exp =
  match exp with
    | True -> True
    | False -> False
    | And(exp0,exp1) ->
      begin
        match (exp0, exp1) with
          | (True,True) -> True
          | (_,False) -> False
          | (False,_) -> False
      end
```

At first glance, this function appears to have the behavior we desire. `True` evaluates to `True`, `False` to `False`, and `True And False` to `False`. This is not, however, a complete implementation.

To find out why, consider the concrete term `True And True And True`. As we have seen before, this translates to the abstract term `And(And(True,True),True)`. When the `eval` function receives that value as its parameter, it matches the value to the case `And` and defines `exp0` and `exp1` as `And(True,True)` and `True`, respectively. We then enter the inner match, and this match fails! None of the terms can match the tuple `(exp0,exp1)` because `exp0` is an entire expression and not just a value, as the match expression is expecting.

Looking back at our attempts to write the `And` rule above, we can see why this `eval` function is flawed: this version of the function does not consider the operands of `And` to be expressions - it expects them to be values. We can see, then, that the `And` clause

in our function is a faithful implementation of *Try 2* of our `And` rule, a rule which we rejected precisely because it could not handle nested expressions.

How can we correct this problem? We are trying to write a faithful implementation of our final `And` rule, which relies on the evaluation of the `And` rule's operands. Thus, in our implementation, we must evaluate those operands; we make this possible by declaring our evaluation function to be recursive.

```
let rec eval exp =
  match exp with
    | True -> True
    | False -> False
    | And(exp0,exp1) ->
      begin
        match (eval exp0, eval exp1) with
          | (True,True) -> True
          | (_,False) -> False
          | (False,_) -> False
      end
```

Observe that, in the above code, we have changed very little. We modified the `eval` function to be recursive. We also added a call to `eval` for each of the operands to the `And` operation. That call alone is sufficient to fix the problem; the process of evaluating those arguments represents the $e_1 \Rightarrow v_1$ and $e_2 \Rightarrow v_2$ preconditions on the `And` rule, while the use of the resultings values in the tuple causes the match to be against v_1 and v_2 rather than e_1 and e_2. The above code is a faithful implementation of the value rule and the `And` rule.

We can now complete the boolean language interpreter by continuing the eval fuction in the same form:

```
let rec eval exp =
  match exp with
    True -> True
  | False -> False
  | Not(exp0) -> (match eval exp0 with
      True -> False
    | False -> True)
  | And(exp0,exp1) -> (match (eval exp0, eval exp1) with
      (True,True) -> True
    | (_,False) -> False
    | (False,_) -> False)

  | Or(exp0,exp1) -> (match (eval exp0, eval exp1) with
      (False,False) -> False
    | (_,True) -> True
    | (True,_) -> True)

  | Implies(exp0,exp1) -> (match (eval exp0, eval exp1) with
      (False,_) -> True
```

```
  | (True,True) -> True
  | (True,False) -> False)
```

The only difference between the operational semantics and the interpreter is that the interpreter is a function. We start with the bottom-left expression in a rule, use the interpreter to recursively produce the value(s) above the line in the rule, and finally compute and return the value below the line in the rule.

Note that the boolean language interpreter above faithfully implements its operational semantics: $e \Rightarrow v$ if and only if `eval`($[\![e]\!]$) returns $[\![v]\!]$ as result. We will go back and forth between these two forms throughout the book. The operational semantics form is used because it is independent of any particular programming language. The interpreter form is useful because we can interpret real programs for nontrivial numbers of steps, something that is difficult to do "on paper" with an operational semantics.

Definition 2.3 (Metacircular Interpreter)**.** *A **metacircular interpreter** is an interpreter for (possibly a subset of) a language x that is written in language x.*

Metacircular interpreters give you some idea of how a language works, but suffer from the non-foundational problems implied in Exercise 2.5. A metacircular interpreter for Lisp (that is, a Lisp interpreter written in Lisp) is a classic programming language theory exercise.

2.3 The F♭ Programming Language

Now that we have seen how to define and understand operational semantics, we will begin to study our first programming language: **F♭**. **F♭** is a shunk (flattened) pure *functional* programming language. [2] It has integers, booleans, and higher-order anonymous functions. In most ways **F♭** is much weaker than OCaml: there are no reals, lists, types, modules, state, or exceptions.

F♭ is untyped, and in this way is it actually more powerful than OCaml. It is possible to write some programs in **F♭** that produce no runtime errors, but which will not typecheck in OCaml. For instance, our encoding of recursion in Section 2.3.5 is not typeable in OCaml. Type systems are discussed in Chapter 6. Because there are no types, runtime errors can occur in **F♭**, such as the application `(5 3)`.

Although very simplistic, **F♭** is still **Turing-complete**. The concept of Turing-completeness has been defined in numerous equivalent ways. One such definition is as follows:

Definition 2.4 (Turing Completeness)**.** *A computational model is **Turing-complete** if every **partial recursive function** can be expressed within it.*

This definition, of course, requires a definition of partial recursive functions (also known as computable functions). Without going into an extensive discussion of foundational material, the following somewhat informal definition will suffice:

[2] Also, any readers familiar with the programming language C♯ as well as basic music theory should find this at least a bit humorous.

Definition 2.5 (Partial Recursive Function)**.** *A function is a partial recursive function if an algorithm exists to calculate it which has the following properties:*

- *The algorithm must have as its input a finite number of arguments.*
- *The algorithm must consist of a finite number of steps.*
- *If the algorithm is given arguments for which the function is defined, it must produce the correct answer within a finite amount of time.*
- *If the algorithm is given arguments for which the function is not defined, it must either produce a clear error or otherwise not terminate. (That is, it must not appear to have produced an incorrect value for the function if no such value is defined.)*

The above definition of a partial recursive function is a mathematical one and thus does not concern itself with execution-specific details such as storage space or practical execution time. No constraints are placed against the amount of memory a computer might need to evaluate the function, the range of the arguments, or that the function terminate before the heat death of the universe (so long as it would eventually terminate for all inputs for which the function is defined).

The practical significance of Turing-completeness is this: there is no computation that could be expressed in another deterministic programming language that cannot be expressed in **F**♭.[3] In fact, **F**♭ is even Turing-complete without numbers or booleans. This language, one with only functions and application, is known as the pure lambda-calculus and is discussed briefly in Section 2.4.4. No deterministic programming language can compute more than the partial recursive functions.

2.3.1 F♭ Syntax

We will take the same approach in defining **F**♭ as we did in defining the boolean language above. We start by describing the grammar of the **F**♭ language to define its **concrete syntax**; the **abstract syntax** is deferred until Section 2.3.7. We can define the grammar of **F**♭ using the following BNF:

[3] This does not guarantee that the **F**♭ representation will be pleasant. Programs written in **F**♭ to perform even fairly simplistic computations such as determining if one number is less than another are excruciating, as we will see shortly.

x	::=	(`a` | `b` | ... | `z`)	*lower-case letters*
		(`A` | `B` | ... | `Z`	*capital letters*
		| `a` | `b` | ... | `z`	*lower-case letters*
		| `0` | `1` | ... | `9`	*digits*
		| `_` | `'` | $\cdots$)$*$	*other characters*
v	::=	x	*variable values*
		| `True` | `False`	*boolean values*
		| `0` | `1` | `-1` | `2` | `-2` | ...	*integer values*
		| `Function` $x \to e$	*function values*
e	::=	v	*value expressions*
		| (`(`e`)`)	*parenthesized expressions*
		| e `And` e | e `Or` e | `Not` e	*boolean expressions*
		| e `+` e | e `-` e | e `=` e |	*numerical expression*
		| e e	*application expression*
		| `If` e `Then` e `Else` e	*conditional expressions*
		| `Let` x `=` e `In` e	*let expression*
		| `Let Rec` f x `=` e `In` e	*recursive let expression*

Note that in accordance with the above BNF, we will be using metavariables e, v, and x to represent expressions, values, and variables respectively. Note the last point: the metavariable x refers to an arbitrary **F♭** variable, not necessarily to the **F♭** variable `x`.

Associativity in **F♭** works in a fashion very similar to OCaml. Function application, for instance, is left associative, meaning that `a b c` has the same meaning as `(a b) c`. As with any language, this associativity is significant in that it affects how source code is parsed into an AST.

2.3.2 Variable Substitution

The main feature of **F♭** is higher-order functions, which also introduces variables. Recall that programs are computed by rewriting them:

```
(Function x -> x + 2)(3 + 2 + 5) ⇒ 12
    because
        3 + 2 + 5 ⇒ 10
            because
                3 + 2 ⇒ 5
            and
                5 + 5 ⇒ 10
    and
        10 + 2 ⇒ 12
```

Note how in this example, the argument is substituted for the variable in the body—this gives us a rewriting interpreter. In other words, **F♭** functions compute by substituting the actual argument for the for parameter; for example,

```
(Function x -> x + 1) 2
```

will compute by substituting 2 for x in the function's body $x+1$, i.e. by computing $2+1$. This is not a very efficient method of computing, but it is a very simple and accurate *description* method, and that is what operational semantics is all about – describing clearly and unambiguously how programs are to compute.

Bound and Free Occurrences of Variables We need to be careful about how variable substitution is defined. For instance,

```
(Function x -> Function x -> x) 3
```

should not evaluate to `Function x -> 3` since the inner `x` is bound by the inner parameter. To correctly formalize this notion, we need to make the following definitions.

Definition 2.6 (Variable Occurrence)**.** *A variable use x* ***occurs*** *in e if x appears somewhere in e. Note we refer only to variable* uses*, not definitions.*

Definition 2.7 (Bound Occurrence)**.** *Any occurrences of variable x in the expression*

`Function` x `->` e

are ***bound****, that is, any free occurrences of x in e are bound occurrences in this expression. Similarly, in the expression*

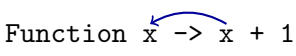
`Let Rec` f x `=` e_1 `In` e_2

occurrences of f and x are bound in e_1 and occurrences of f are bound in e_2. Note that x is not bound in e_2, but only in e_1, the body of the function.

Definition 2.8 (Free Occurrence)**.** *A variable x occurs* ***free*** *in e if it has an occurrence in e which is not a bound occurrence.*

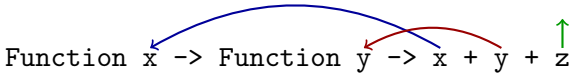
Let's look at a few examples of bound versus free variable occurrences.

Example 2.6.



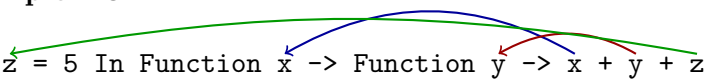
`x` is bound in the body of this function.

Example 2.7.



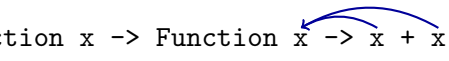
`x` and `y` are bound in the body of this function. `z` is free.

Example 2.8.



`x`, `y`, and `z` are all bound in the body of this function. `x` and `y` are bound by their respective function declarations, and `z` is bound by the `Let` statement. Note that, while **F♭** contains `Let` as syntax, it can be defined as a macro (see Section 2.3.4 below). Binding rules work similarly for `Functions` and `Let` statements.

Example 2.9.



`x` is bound in the body of this function. Note that both `x` usages are bound to the inner variable `x`.

Definition 2.9 (Closed Expression)**.** *An expression e is closed if it contains no free variable occurrences. All programs we execute are closed (no link-time errors) – non-closed programs don't diverge, we can't even contemplate executing them because they are not in the domain of the evaluation relation.*

Of the examples above, Examples 2.6, 2.8, and 2.9 are closed expressions. Example 2.7 is not a closed expression.

Now that we have an understanding of bound and free variables, we can give a formal definition of variable substitution.

Definition 2.10 (Variable Substitution)**.** *The variable substitution of x for e' in e, denoted $e[e'/x]$, is the expression resulting from the operation of replacing all free occurrences of x in e with e'. For now, we assume that e' is a closed expression.*

Here is an equivalent inductive definition of substitution:

$$
\begin{array}{rcll}
x[v/x] & = & v & \\
x'[v/x] & = & x' & x \neq x' \\
(\texttt{Function}\ x \to e)[v/x] & = & (\texttt{Function}\ x \to e) & \\
(\texttt{Function}\ x' \to e)[v/x] & = & (\texttt{Function}\ x' \to e[v/x]) & x \neq x' \\
(\texttt{Let}\ x\ \texttt{=}\ e_1\ \texttt{In}\ e_2)[v/x] & = & \texttt{Let}\ x\ \texttt{=}\ e_1[v/x]\ \texttt{In}\ e_2 & \\
(\texttt{Let}\ x'\ \texttt{=}\ e_1\ \texttt{In}\ e_2)[v/x] & = & \texttt{Let}\ x'\ \texttt{=}\ e_1[v/x]\ \texttt{In}\ e_2[v/x] & x \neq x' \\
n[v/x] & = & n \text{ for } n \in \mathbb{Z} & \\
\texttt{True}[v/x] & = & \texttt{True} & \\
\texttt{False}[v/x] & = & \texttt{False} & \\
(e_1 + e_2)[v/x] & = & e_1[v/x] + e_2[v/x] & \\
(e_1\ \texttt{And}\ e_2)[v/x] & = & e_1[v/x]\ \texttt{And}\ e_2[v/x] & \\
 & \vdots & &
\end{array}
$$

For example, let us consider a simple application of a function: `(Function x -> x + 1) 2`. We know that, to evaluate this function, we simply replace all instances of `x` in the body with `2`. This is written `(x + 1)[2/x]`. Given the above definition, we can conclude that the result must be `3`.

While this may not seem like an illuminating realization, the fact that this is mathematically discernable gives us a starting point for more complex subsitutions. Consider the following example.

Example 2.10.

Expression:

```
(Function x -> Function y -> (x + x + y)) 5
```

Substitution:

```
(Function y -> (x + x + y))[5/x]
= (Function y -> (x + x + y)[5/x])
= Function y -> (x[5/x] + x[5/x] + y[5/x])
= Function y -> (5 + 5 + y)
```

α-conversion

In Example 2.9, we saw that it is possible for two variables to share the same name. The variables themselves are, of course, distinct and follow the same rules of scope in **F♭** as they do in OCaml. But reading expressions which make frequent use of the same variable name for different variables can be very disorienting. For example, consider the following expression.

```
Let Rec f x =
    If x = 1 Then
        (Function f -> f (x - 1)) (Function x -> x)
    Else
        f (x - 1)
In f 100
```

How does this expression evaluate? It is a bit difficult to tell simply by looking at it because of the tricky bindings. We can make it much easier to understand by using different names. α-conversion is the process of replacing a variable definition and all occurrences bound to it with a variable of a different name.

Example 2.11.

`Function x -> x + 1`

becomes

`Function z -> z + 1`

Example 2.11 shows a simple case in which `x` is substituted for `z`. For cases in which the same variable name is used numerous times, we can use the same approach. Consider Example 2.12 in which the inner variable `x` is α-converted to `z`.

Example 2.12.

`Function x -> Function x -> x`

becomes

`Function x -> Function z -> z`

Similarly, we could rename the outer variable to `z` as shown in Example 2.13. Note that in this case, the occurrence of `x` is *not* changed, as it is bound by the inner variable and not the outer one.

Example 2.13.
`Function (x) -> Function x -> x`
becomes
`Function (z) -> Function x -> x`

Let's figure out what variable occurrences are bound to which function in our previous confusing function and rewrite the function in a clearer way by using α-conversion. One possible result is as follows:

```
Let Rec f x =
   If x = 1 Then
      (Function z -> z (x - 1)) (Function y -> y)
   Else
      f (x - 1)
In f 100
```

Now it's much easier to understand what is happening. If the function `f` is applied to an integer which is not `1`, it merely applies itself again to the argument which is one less than the one it received. Since we are evaluating `f 100`, that results in `f 99`, which in turn evaluates `f 98`, and so on. Eventually, `f 1` is evaluted.

When `f 1` is evaluated, we explore the other branch of the `If` expression. We know that `x` is 1 at this point, so we can see that the evaluated expression is `(Function z -> z 0) (Function y -> y)`. Using substitution gives us `(Function y -> y) 0`, which in turn gives us `0`. So we can conclude that the expression above will evaluate to `0`.

Observe, however, that we did not formally prove this; so far, we have been treating substitution and other operations in a cavalier fashion. In order to create a formal proof, we need a set of operational semantics which dictates how evaluation works in **F♭**. Section 2.3.3 walks through the process of creating an operational semantics for the **F♭** language and gives us the tools needed to prove what we concluded above.

2.3.3 Operational Semantics for F♭

We are now ready to begin defining operational semantics for **F♭**. For the same reasons as in our boolean language, we will need a rule which relates values to values in $\Rightarrow$:

(Value Rule) $\dfrac{}{v \Rightarrow v}$

We can also define boolean operations for **F♭** in the same way as we did for the boolean language above. Note, however, that not all values in **F♭** are booleans. Fortunately, our definition of the rules addresses this for us, as there is (for example) no logical and of the values `5` and `3`. That is, we know that these rule only apply to **F♭** boolean values because they use operations which are only defined for **F♭** boolean values.

(Not Rule) $$\frac{e \Rightarrow v}{\texttt{Not } e \Rightarrow \text{the negation of } v}$$

(And Rule) $$\frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2}{e_1 \texttt{ And } e_2 \Rightarrow \text{the logical and of } v_1 \text{ and } v_2}$$

$\vdots$

We can also define operations over integers in much the same way. For sake of clarity, we will explicitly restrict these rules such that they operate only on expressions which evaluate to integers.

(+ Rule) $$\frac{e_1 \Rightarrow v_1, \quad e_2 \Rightarrow v_2 \text{ where } v_1, v_2 \in \mathbb{Z}}{e_1 \texttt{ + } e_2 \Rightarrow \text{the integer sum of } v_1 \text{ and } v_2}$$

(- Rule) $$\frac{e_1 \Rightarrow v_1, \quad e_2 \Rightarrow v_2 \text{ where } v_1, v_2 \in \mathbb{Z}}{e_1 \texttt{ - } e_2 \Rightarrow \text{the integer difference of } v_1 \text{ and } v_2}$$

As with the boolean rules, observe that these rules allow the $\Rightarrow$ relation to be applied recursively: `5 + (4 - 3)` can be evaluated using the `+` rule because `4 - 3` can be evaluated using the `-` rule first.

These rules allow us to write **F♭** programs containing boolean expressions or **F♭** programs containing integer expressions, but we currently have no way to combine the two. There are two mechanisms we use to mix the two in a program: conditional expressions and comparison operators. The only comparison operator in **F♭** is the `=` operator, which compares the values of integers. We define the `=` rule as follows.

(= Rule) $$\frac{e_1 \Rightarrow v_1, \quad e_2 \Rightarrow v_2 \text{ where } v_1, v_2 \in \mathbb{Z}}{e_1 \texttt{ = } e_2 \Rightarrow \texttt{True} \text{ if } v_1 \text{ and } v_2 \text{ are identical, else } \texttt{False}}$$

Note that the `=` rule is defined only where v_1 and v_2 are integers. Due to this constraint, the expression `True = True` is not evaluatable in **F♭**. This is, of course, a matter of choice; as a language designer, one may choose to remove that constraint and allow boolean values to be compared directly. To formalize this, however, a change to the rules would be required. A faithful implementation of **F♭** using the above `=` rule is *required* to reject the expression `True = True`.

An intuitive definition of a conditional expression is that it evalutes to the value of one expression if the boolean is true and the value of the other expression if the boolean is false. While this is true, the particulars of how this is expressed in the rule are vital. Let us consider the following flawed attempt at a conditional expression rule:

(Flawed If Rule) $$\frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2 \quad e_3 \Rightarrow v_3}{\texttt{If } e_1 \texttt{ Then } e_2 \texttt{ Else } e_3 \Rightarrow v_2 \text{ if } v_1 \text{ is } \texttt{True}, v_3 \text{ otherwise}}$$

It seems that this rule allows us to evaluate many of the conditional expressions we want to evaluate. But let us consider this expression:

```
If True Then 0 Else (True + True)
```

If we attempted to apply the above rule to the previous expression, we would find that the precondition $e_3 \Rightarrow v_3$ would not hold; there is no rule which can relate `True + True` $\Rightarrow v$ for any v since the `+` rule only applies to integers. Nonetheless, we want the expression to evaluate to `0`. So how can we achieve this result?

In this case, we have no choice but to write two different rules with distinct preconditions. We can capture all of the relationships in the previous rule and yet allow expressions such as the previous to evaluate by using the following two rules:

(`If True` Rule)
$$\frac{e_1 \Rightarrow \texttt{True}, \quad e_2 \Rightarrow v_2}{\texttt{If } e_1 \texttt{ Then } e_2 \texttt{ Else } e_3 \Rightarrow v_2}$$

(`If False` Rule)
$$\frac{e_1 \Rightarrow \texttt{False}, \quad e_3 \Rightarrow v_3}{\texttt{If } e_1 \texttt{ Then } e_2 \texttt{ Else } e_3 \Rightarrow v_3}$$

Again, the key difference between these two rules is that they have different sets of preconditions. Note that the `If True` rule does not evaluate e_3, nor does the `If False` rule evaluate e_2. This allows the untraveled logic paths to contain unevaluatable expressions without necessarily preventing the expression containing them from evaluating.

Application

We are now ready to approach one of the most difficult **F♭** rules: application. How can we formalize the evaluation of an expression like `(Function x -> x + 1) (5 + 2)`? We saw in Section 2.3.2 that we can evaluate a function application by using variable substitution. As we have a mathematical definition for the substitution operation, we can base our function application rule around it.

Suppose we wish to evaluate the above expression. We can view application in two parts: the function being applied and the argument to the function. We wish to know to what the expression evaluates; thus, we are trying to establish that $e_1 e_2 \Rightarrow v$ for some v.

(Application Rule (Part 1))
$$\frac{?}{e_1\ e_2 \Rightarrow v}$$

In our boolean operations, we needed to evaluate the arguments before attempting an operation over them (in order to allow recursive expressions). The same is true of our application rule; in `(Function x -> x + 1) (5 + 2)`, we must evaluate `5 + 2` before it can be used as an argument.[4] We must do likewise with the function we are applying.

[4] Actually, some languages would perform substitution before evaluating the expression, but **F♭** and most traditional languages do not. Discussion of this approach is handled in Section 2.3.6.

(Application Rule (Part 2)) $$\frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2 \quad ?}{e_1\ e_2 \Rightarrow v}$$

We obviously aren't finished, though, as we still don't have any preconditions which allow us to relate v to something. Additionally, we know we will need to use variable substitution, but we have no metavariables representing **F♭** variables in the above rule. We can fix this by reconsidering how we evaluate the first argument; we know that the application rule only works when applying *functions*. In restricting our rule to applying functions, we can name some metavariables to describe the function's contents.

(Application Rule (Part 3)) $$\frac{e_1 \Rightarrow \texttt{Function}\ x\ \texttt{->}\ e \quad e_2 \Rightarrow v_2 \quad ?}{e_1\ e_2 \Rightarrow v}$$

In the above rule, x is the metavariable representing the function's variable while e represents the function's body. Now that we have this information, we can define function application in terms of variable substitution. When we apply `Function x -> x + 1` to a value such as `7`, we wish to replace all instances of `x`, the function's variable, in the function's body with `7`, the provided argument. Formally,

(Application Rule) $$\frac{e_1 \Rightarrow \texttt{Function}\ x\ \texttt{->}\ e \quad e_2 \Rightarrow v_2 \quad e[v_2/x] \Rightarrow v}{e_1\ e_2 \Rightarrow v}$$

F♭ Recursion

We now have a very complete set of rules for the **F♭** language. We do not, however, have a rule for `Let Rec`. As we will see, `Let Rec` is not actually necessary in basic **F♭**; it is possible to build recursion out of the rules we already have. Later, however, we will create variants of **F♭** with type systems in which it will be impossible for that approach to recursion to work. For that reason as well as our immediate convenience, we will define the `Let Rec` rule now.

Again, we start with an iterative approach. We know that we want to be able to evaluate the `Let Rec` expression, so we provide metavariables to represent the components of the expression itself.

(Recursive Application Rule (Part 1)) $$\frac{?}{\texttt{Let Rec}\ f\ x\ \texttt{=}\ e_1\ \texttt{In}\ e_2 \Rightarrow v}$$

Let us consider what we wish to accomplish. Consider for a moment a recursive approach to the summation of the numbers between 1 and 5:

```
Let Rec f x =
   If x = 1 Then
      1
   Else
      f (x - 1) + x
In f 5
```

If we focus on the last line (`In f 5`), we can see that we want the body of the recursive function to be applied to the value `5`. We can write our rule accordingly by replacing `f` with the function's body. We must make sure to use the same metavariable to represent the function's argument in order to allow the new function body's variable to be captured. We reach the following rule.

(Recursive Application Rule (Part 2))

$$\frac{e_2[\texttt{(Function } x \texttt{ -> } e_1\texttt{)}/f] \Rightarrow v}{\texttt{Let Rec } f\ x \texttt{ = } e_1 \texttt{ In } e_2 \Rightarrow v}$$

We can test our new rule by applying it to our recursive summation above.

$$\dfrac{\dfrac{\dfrac{}{\texttt{Function x ->}\cdots \Rightarrow \texttt{Function x ->}\cdots} \quad \dfrac{}{\texttt{5} \Rightarrow \texttt{5}} \quad \dfrac{\dfrac{}{\texttt{5 = 1} \Rightarrow \texttt{False}} \quad \dfrac{\dfrac{???}{\texttt{f (5-1)} \Rightarrow v'} \quad \dfrac{}{\texttt{5} \Rightarrow \texttt{5}}}{\texttt{f (5-1) + 5} \Rightarrow v}}{\texttt{If 5 = 1 Then 1 Else f (5-1) + 5} \Rightarrow v}}{\texttt{(Function x -> If x = 1 Then 1 Else f (x-1) + x) 5} \Rightarrow v}}{\texttt{Let Rec f x = If x = 1 Then 1 Else f (x-1) + x In f 5} \Rightarrow v}$$

As foreshadowed by its label above, our recursion rule is not yet complete. When we reach the evaluation of `f (5-1)`, we are at a loss; `f` is not bound. Without a binding for `f`, we have no way of repeating the recursion.

In addition to replacing the invocation of f with function application in e_2, we need to ensure that any occurrences of f in the body of the function itself are bound. To what do we bind them? We could try to replace them with function applications as well, but this leads us down a rabbit hole; each function application would require yet another replacement. We can, however, replace applications of f in e_1 with *recursive* applications of f by reintroducing the `Let Rec` syntax. This leads us to the following application rule:

(Recursive Application Rule (Part 3))

$$\frac{e_2[\texttt{Function } x \texttt{ -> } e_1[\texttt{(Let Rec } f\ x \texttt{ = } e_1 \texttt{ In } f\texttt{)}/f]/f] \Rightarrow v}{\texttt{Let Rec } f\ x \texttt{ = } e_1 \texttt{ In } e_2 \Rightarrow v}$$

While this makes a certain measure of sense, it isn't quite correct. In Section 2.3.2, we saw that substitution must replace a variable with a *value*, while the `Let Rec` term

above is an expression. Fortunately, we have functions as values; thus, we can put the expression inside of a function and ensure that we call it with the appropriate argument.

(Recursive Application Rule)

$$\frac{e_2[\texttt{Function } x \texttt{ -> } e_1[(\texttt{Function } x \texttt{ -> Let Rec } f\ x \texttt{ = } e_1 \texttt{ In } f\ x)/f]/f] \Rightarrow v}{\texttt{Let Rec } f\ x \texttt{ = } e_1 \texttt{ In } e_2 \Rightarrow v}$$

Now, instead of encountering `f (5-1)` when we evaluate the summation example, we encounter `Let Rec f x = If x = 1 Then 1 Else f (x-1) + x In f (5-1)`. This allows us to recurse back into the `Let Rec` rule. Eventually, we may reach a branch which does not evaluate the `Else` side of the conditional expression, in which case that `Let Rec` is not expanded (allowing us to terminate). Each application of the rule effectively "unrolls" one level of recursion.

In summary, we have the following operational semantics for $\mathbf{F}\flat$ (excluding some repetetive rules such as the `-` rule):

(Value Rule) $$\frac{}{v \Rightarrow v}$$

(Not Rule) $$\frac{e \Rightarrow v}{\texttt{Not } e \Rightarrow \text{the negation of } v}$$

(And Rule) $$\frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2}{e_1 \texttt{ And } e_2 \Rightarrow \text{the logical and of } v_1 \text{ and } v_2}$$

(`+` Rule) $$\frac{e_1 \Rightarrow v_1, \quad e_2 \Rightarrow v_2 \text{ where } v_1, v_2 \in \mathbb{Z}}{e_1 \texttt{ + } e_2 \Rightarrow \text{the integer sum of } v_1 \text{ and } v_2}$$

(`=` Rule) $$\frac{e_1 \Rightarrow v_1, \quad e_2 \Rightarrow v_2 \text{ where } v_1, v_2 \in \mathbb{Z}}{e_1 \texttt{ = } e_2 \Rightarrow \texttt{True} \text{ if } v_1 \text{ and } v_2 \text{ are identical, else } \texttt{False}}$$

(`If True` Rule) $$\frac{e_1 \Rightarrow \texttt{True}, \quad e_2 \Rightarrow v_2}{\texttt{If } e_1 \texttt{ Then } e_2 \texttt{ Else } e_3 \Rightarrow v_2}$$

(`If False` Rule) $$\frac{e_1 \Rightarrow \texttt{False}, \quad e_3 \Rightarrow v_3}{\texttt{If } e_1 \texttt{ Then } e_2 \texttt{ Else } e_3 \Rightarrow v_3}$$

(Application Rule) $$\frac{e_1 \Rightarrow \texttt{Function } x \texttt{ -> } e, \quad e_2 \Rightarrow v_2, \quad e[v_2/x] \Rightarrow v}{e_1\ e_2 \Rightarrow v}$$

(Let Rule) $$\frac{e_1 \Rightarrow v_1 \quad e_2[v_1/x] \Rightarrow v_2}{\texttt{Let } x \texttt{ = } e_1 \texttt{ In } e_2 \Rightarrow v_2}$$

(Let Rec)

$$\frac{e_2[\texttt{Function } x \texttt{ -> } e_1[(\texttt{Function } x \texttt{ -> Let Rec } f\ x \texttt{ = } e_1 \texttt{ In } f\ x)/f]/f] \Rightarrow v}{\texttt{Let Rec } f\ x \texttt{ = } e_1 \texttt{ In } e_2 \Rightarrow v}$$

Let us consider a few examples of proof trees using the **F♭** operational semantics.

Example 2.14.

Expression:
```
If 3 = 4 Then 5 Else 4 + 2
```

Proof:

$$\dfrac{\dfrac{\dfrac{}{\texttt{3} \Rightarrow \texttt{3}} \quad \dfrac{}{\texttt{4} \Rightarrow \texttt{4}}}{\texttt{3 = 4} \Rightarrow \texttt{False}} \quad \dfrac{\dfrac{}{\texttt{4} \Rightarrow \texttt{4}} \quad \dfrac{}{\texttt{2} \Rightarrow \texttt{2}}}{\texttt{4 + 2} \Rightarrow \texttt{6}}}{\texttt{If 3 = 4 Then 5 Else 4 + 2} \Rightarrow \texttt{6}}$$

Example 2.15.

Expression:
```
(Function x -> If 3 = x Then 5 Else x + 2) 4
```

Proof:

$$\dfrac{\dfrac{}{\texttt{Function x ->}\cdots \Rightarrow \texttt{Function x ->}\cdots} \quad \dfrac{}{\texttt{4} \Rightarrow \texttt{4}} \quad \dfrac{\texttt{If 3 = 4 Then 5 Else 4 + 2} \Rightarrow \texttt{6}}{\text{by Example 2.14}}}{\texttt{(Function x -> If 3 = x Then 5 Else x + 2) 4} \Rightarrow \texttt{6}}$$

Example 2.16.

Expression:
```
(Function f -> Function x -> f(f x))(Function y -> y - 1) 4
```

Proof:

Due to the size of the proof, it is broken into multiple parts. We use $v \Rightarrow \bigstar$ as an abbreviation for $v \Rightarrow v$ (when v is lengthy) for brevity.

Part 1:

$$\dfrac{\dfrac{}{\texttt{Function f -> Function x -> f(f x)} \Rightarrow \bigstar} \quad \dfrac{}{\texttt{Function y -> y - 1} \Rightarrow \bigstar} \quad \dfrac{}{\texttt{(Function x -> (Function y -> y - 1) ((Function y -> y - 1) x))} \Rightarrow \bigstar}}{\texttt{(Function f -> Function x -> f(f x))(Function y -> y - 1)} \Rightarrow \texttt{(Function x -> (Function y -> y - 1) ((Function y -> y - 1) x))}}$$

Part 2:

$$\dfrac{\dfrac{}{\textit{(by part 1)}} \quad \dfrac{}{\texttt{4} \Rightarrow \texttt{4}} \quad \dfrac{\dfrac{}{\texttt{Function y -> y - 1} \Rightarrow \bigstar} \quad \dfrac{\dfrac{\dfrac{}{\texttt{Function y -> y - 1} \Rightarrow \bigstar} \quad \dfrac{}{\texttt{4} \Rightarrow \texttt{4}} \quad \dfrac{\dfrac{}{\texttt{4} \Rightarrow \texttt{4}} \quad \dfrac{}{\texttt{1} \Rightarrow \texttt{1}}}{\texttt{4 - 1} \Rightarrow \texttt{3}}}{\texttt{((Function y -> y - 1) 4)} \Rightarrow \texttt{3}} \quad \dfrac{\dfrac{}{\texttt{3} \Rightarrow \texttt{3}} \quad \dfrac{}{\texttt{1} \Rightarrow \texttt{1}}}{\texttt{3 - 1} \Rightarrow \texttt{2}}}{\texttt{(Function y -> y - 1) ((Function y -> y - 1) 4)} \Rightarrow \texttt{2}}}{\texttt{(Function f -> Function x -> f(f(x)))(Function y -> y - 1) 4} \Rightarrow \texttt{2}}$$

Interact with F♭. Tracing through recursive evaluations is difficult, and therefore the reader should invest some time in exploring the semantics of `Let Rec`. A good way to do this is by using the **F♭** interpreter. Try evaluating the expression we looked at above:

```
# Let Rec f x =
    If x = 1 Then 1 Else x + f (x - 1)
  In f 3;;
==> 6
```

Another interesting experiment is to evaluate a recursive function without applying it. Notice that the result is equivalent to a single application of the `Let Rec` rule. This is a good way to see how the "unwrapping" actually takes place:

```
# Let Rec f x =
    If x = 1 Then 1 Else x + f (x - 1)
  In f;;
==> Function x ->
      If x = 1 Then
        1
      Else
        x + (Let Rec f x =
          If x = 1 Then
            1
          Else
            x + (f) (x - 1)
        In
          f) (x - 1)
```

As we mentioned before, one of the main benefits of defining an operational semantics for a language is that we can rigorously verify claims about that language. Now that we have defined the operational semantics for **F♭**, we can prove a few things about it.

Lemma 2.3. ***F♭*** *is deterministic.*

Proof. By inspection of the rules, at most one rule can apply at any time. □

Lemma 2.4. ***F♭*** *is* not *normalizing.*

Proof. To show that a language is not normalizing, we simply show that there is some e such that there is no v with $e \Rightarrow v$. Let e be `(Function x -> x x)(Function x -> x x)`. $e \not\Rightarrow v$ for any v. Thus, **F♭** is not normalizing. □

The expression in this proof is a very interesting one which we will examine in more detail in Section 2.3.5. It does not evaluate to a value because each step in its evaluation produces itself as a precondition. This is roughly analogous to trying to prove proposition A by using A as a given.

In this case, the expression does not evaluate because it never runs out of work to do. This is not the only kind of non-normalizing expression which appears in **F♭**; another kind consists of those expressions for which no evaluation rule applies. For example, `(4 3)` is a simpler expression that is non-normalizing. No rule exists to evaluate $(e_1\ e_2)$ when e_1 is not a function expression.

Both of these cases look like divergence as far as the operational semantics are concerned. In the case of an interpreter, however, these two kinds of expressions behave differently. The expression which never runs out of work to do typically causes an interpreter to loop infinitely (as most interpreters are not clever enough to realize that they will never finish). The expressions which attempt application to non-functions, on the other hand, cause an interpreter to provide a clear error in the form of an exception. This is because the error here is easily detectable; the interpreter, in attempting to evaluate these expressions, can quickly discover that none of its available rules apply to the expression and respond by raising an exception to that effect. Nonetheless, both are theoretically equivalent.

2.3.4 The Expressiveness of F♭

F♭ doesn't have many features, but it is possible to do much more than it may seem. As we said before, **F♭** is Turing complete, which means, among other things, that any OCaml operation may be encoded in **F♭**. We can informally represent encodings in our interpreter as macros using OCaml `let` statements. A macro is equivalent to a statement like "let F be `Function x -> ...`"

Logical Combinators First, there are the classic **logical combinators**, simple functions for recombining data.

```
combI = Function x -> x
combK = Function x -> Function y -> x
combS = Function x -> Function y -> Function z -> (x z) (y z)
combD = Function x -> x x
```

Tuples Tuples and lists are encodable from just functions, and so they are not needed as primitives in a language. Of course for an *efficient* implementation you would want them to be primitives; thus doing this encoding is simply an exercise to better understand the nature of functions and tuples. We will define a 2-tuple (pairing) constructor; From a pair you can get a n-tuple by building it from pairs. For example, $(1, (2, (3, 4)))$ represents the 4-tuple $(1, 2, 3, 4)$.

First, we need to define a pair constructor, `pr`. A first (i.e., slightly buggy) approximation of the constructor is as follows.

`pr (`e_1`, `e_2`)` $\stackrel{\text{def}}{=}$ `Function x -> x` e_1 e_2

We use the notation $a \stackrel{\text{def}}{=} b$ to indicate that a is an abbreviation for b. For example, we might have a problem in which the incrementer function is commonly used; it would make sense, then, to define `inc` $\stackrel{\text{def}}{=}$ `Function x -> x + 1`. Note that such abbreviations do not change the underlying meaning of the expression; it is simply for convenience. The same concept applies to macro definitions in programming languages. By creating a new macro, one is not changing the math behind the programming language; one is merely defining a more terse means of expressing a concept.

Based on the previous definition of `pr`, we can define the following macros for projection:

`left (`e`)` $\stackrel{\text{def}}{=}$ e `(Function x -> Function y -> x)`
`right (`e`)` $\stackrel{\text{def}}{=}$ e `(Function x -> Function y -> y)`

Now let's take a look at what's happening. `pr` takes a left expression, e_1, and a right expression, e_2, and packages them into a function that applies its argument `x` to e_1 and

e_2. Because functions are values, the result won't be evaluated any further, and e_1 and e_2 will be packed away in the body of the function until it is applied. Thus `pr` succeeds in "storing" e_1 and e_2.

All we need now is a way to get them out. For that, look at how the projection operations `left` and `right` are defined. They're both very similar, so let's concentrate only on the projection of the left element. `left` takes one of our pairs, which is encoded as a function, and applies it to a curried function that returns its first, or leftmost, element. Recall that the pair itself is just a function that applies its argument to e_1 and e_2. So when the curried `left` function that was passed in is applied to e_1 and e_2, the result is e_1, which is exactly what we want. `right` is similar, except that the curried function returns its second, or rightmost, argument.

Before we go any further, there is a technical problem involving our encoding of `pr`. Suppose e_1 or e_2 contain a free occurrence of `x` when `pr` is applied. Because `pr` is defined as `Function x -> x` e_1 e_2, any free occurrence `x` contained in e_1 or e_2 will become bound by `x` after `pr` is applied. This is known as variable **capture**. To deal with capture here, we need to change our definition of `pr` to the following.

$$\texttt{pr (}e_1\texttt{, }e_2\texttt{)} \stackrel{\text{def}}{=} \texttt{(Function e1 -> Function e2 -> Function x -> x e1 e2)}\ e_1\ e_2$$

This way, instead of textually substituting for e_1 and e_2 directly, we pass them in as functions. This allows the interpreter evaluate e_1 and e_2 to values before passing them in, and also ensures that e_1 and e_2 are closed expressions. This eliminates the capture problem, because any occurrence of `x` is either bound by a function declaration *inside* e_1 or e_2, or was bound outside the entire `pr` expression, in which case it must have already been replaced with a value at the time that the `pr` subexpression is evaluated. Variable capture is an annoying problem that we will see again in Section 2.4.

Now that we have polished our definitions, let's look at an example of how to use these encodings. First, let's create the pair `p` as $(4, 5)$.

$$p \stackrel{\text{def}}{=} \texttt{pr (4, 5)} \Rightarrow \texttt{Function x -> x 4 5}$$

Now, let's project the left element of p.

$$\texttt{left}\ p \equiv \texttt{(Function x -> x 4 5) (Function x -> Function y -> x)}$$

We use the notation $a \equiv b$ to indicate the expansion of a specific macro instance. In this case, `left` p expands to what we see above, which then becomes

$$\texttt{(Function x -> Function y -> x) 4 5} \Rightarrow \texttt{4}.$$

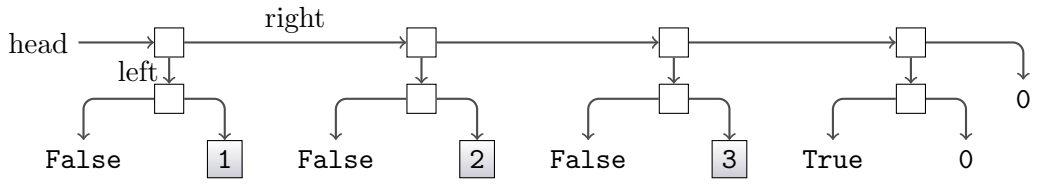


Figure 2.1: Lists Implemented Via Pairs

This encoding works, and has all the expressiveness of real tuples. There are, nonetheless, a few problems with it. First of all, consider

`left (Function x -> 0)` $\Rightarrow$ `0`.

We really want the interpreter to produce a run-time error here, because a function is not a pair.

Similarly, suppose we wrote the program `right(pr(3, pr(4, 5)))`. One would expect this expression to evaluate to `pr(4, 5)`, but remember that pairs are not values in our language, but simply encodings, or macros. So in fact, the result of the computation is `Function x -> x 4 5`. We can only guess that this is intended to be a pair. In this respect, the encoding is flawed, and we will, in Chapter 3, introduce "real" n-tuples into an extension of **F**♭ to alleviate these kinds of problems.

Lists Lists can also be implemented via pairs. In fact, pairs of pairs are technically needed because we need a flag to mark the end of list. The list `[1; 2; 3]` is represented by `pr (pr(false,1), pr (pr(false,2), pr (pr(false,3), emptylist)))` where `emptylist` $\stackrel{\text{def}}{=}$ `pr(pr(true,0),0)`. The true/false flag is used to mark the end of the list: only the empty list is flagged true. The implementation is as follows.

```
cons (x, y) ≝ pr(pr(Bool false, x), y)
emptylist ≝ pr(pr(Bool true, Int 0),Int 0)
head x ≝ right(left x)
tail x ≝ right x
isempty l ≝ (left (left l))
length ≝ Let Rec len x =
   If isempty(x) Then 0 Else 1 + len (tail x) In len
```

In addition to tuples and lists, there are several other concepts from OCaml that we can encode in **F**♭. We review a few of these encodings below. For brevity and readability, we will switch back to the concrete syntax.

Functions with Multiple Arguments Functions with multiple arguments are done with currying just as in OCaml. For example

```
Function x -> Function y -> x + y
```

The Let Operation Although **F♭** includes syntax for `Let`, it is quite simple to encode:

(`Let` x `=` e `In` e') $\stackrel{\text{def}}{=}$ (`Function` x `->` e') e

For example,

(`Let x = 3 + 2 In x + x`) $\equiv$ (`Function x -> x + x`) (`3 + 2`) $\Rightarrow 10$.

Sequencing Notice that **F♭** has no sequencing (`;`) operation. Because **F♭** is a pure functional language, writing e`;` e' is really just equivalent to writing e', since e will never get used. Hence, sequencing really only has meaning in languages with side-effects. Nonetheless, sequencing is definable in the following manner.

e`;` $e' \stackrel{\text{def}}{=}$ (`Function` n `->` e') e,

where n is chosen so as not to be free in e'. This will first execute e, throw away the value, and then execute e', returning its result as the final result of e`;` e'.

Freezing and Thawing We can stop and re-start computation at will by freezing and thawing.

`Freeze` $e \stackrel{\text{def}}{=}$ `Function` n `->` e
`Thaw` $e \stackrel{\text{def}}{=} e$ `0`

We need to make sure that n is a fresh variable so that it is not free in e. Note that the `0` in the application could be any value. `Freeze` e freezes e, keeping it from being computed. `Thaw` e starts up a frozen computation. As an example,

```
Let x = Freeze (2 + 3) In (Thaw x) + (Thaw x)
```

This expression has same value as the equivalent expression without the freeze and thaw, but the `2 + 3` is evaluated twice. Again, in a pure functional language the only difference is that freezing and thawing is less efficient. In a language with side-effects, if the frozen expression causes a side-effect, then the freeze/thaw version of the function may produce results different from those of the original function, since the frozen side-effects will be applied as many times as they are thawed.

2.3.5 Russell's Paradox and Encoding Recursion

F♭ has a built-in `Let Rec` operation to aid in writing recursive functions, but its actually not needed because recursion is definable in **F♭**. The encoding is a non-obvious one, and so before we introduce it, we present some background information. As we will see, the encoding of recursion is closely related to a famous set-theoretical paradox due to Russell.

Let us begin by posing the following question. *How can programs compute forever in **F♭** without recursion?* The answer to this question comes in the form of a seemingly simple expression:

```
(Function x -> x x)(Function x -> x x)
```

Recall from Lemma 2.2, that a corollary to the existence of this expression is that **F♭** is not normalizing. This computation is odd in some sense. `(x x)` is a function being applied to itself. There is a logical paradox at the heart of this non-normalizing computation, namely Russell's Paradox.

Russell's Paradox

In Frege's set theory (circa 1900), sets were written as predicates $P(x)$. We can view predicates as single-argument functions which return a boolean value: true if the argument is in the set represented by the predicate and false if it is not. Testing membership in a set is done via application of the predicate. For example, consider the predicate

```
Function x -> (x = 2 Or x = 3 Or x = 5)
```

This predicate represents the integer set $\{2, 3, 5\}$ since it will return `True` for any of the elements in that set and `False` for all other arguments. If we were to extend **F♭** to include a native integer less-than operator, the predicate

```
Function x -> x < 2
```

would represent an infinitely-sized set containing all integer values less than `2` (as **F♭** still has no notion of real numbers). In general, given a predicate P representing a set S,

$e \in S$ iff $P\ e \Rightarrow$ `True`

Russell discovered a paradox in Frege's set theory, and it can be expressed in the following way.

Definition 2.11 (Russell's Paradox)**.** *Let P be the set of all sets that do not contain themselves as members. Is P a member of P?*

Asking whether or not a set is a member of itself seems like strange question, but in fact there are many sets that are members of themselves. The infinitely receding set $\{\{\{\{\ldots\}\}\}\}$ has itself as a member. The set of things that are not apples is also a member of itself (clearly, a set of non-apples is not an apple). These kinds of sets arise only in "non-well-founded" set theory.

To explore the nature of Russell's Paradox, let us try to answer the question it poses: Does P contain itself as a member? Suppose the answer is yes, and P does contain itself as a member. If that were the case then P should not be in P, which is the set of all sets that do *not* contain themselves as members. Suppose, then, that the answer is no, and that P does not contain itself as a member. Then P should have been included in P, since it doesn't contain itself. In other words, P is a member of P if and only if it isn't. Hence Russell's Paradox is indeed a paradox.

This can also be illustrated by using **F♭** functions as predicates. Specifically, we will write a predicate for P above. We must define P to accept an argument (which we know to be a set - a predicate in our model) and determine if it contains itself (pass it to itself as an argument). Thus, our representation of P is `Function x -> Not(x x)`. We merely apply P to itself to get our answer.

Definition 2.12 (Computational Russell's Paradox)**.** *Let*

$P \stackrel{def}{=}$ *`Function x -> Not(x x).`*

What is the result of $P\ P$? Namely, what is

`(Function x -> Not(x x)) (Function x -> Not(x x))`?

If this **F♭** program were evaluated, it would run forever. We can informally detect the pattern just by looking at a few passes of an evaluation proof:

$$\dfrac{\dfrac{\dfrac{\vdots}{\texttt{Not (Not ((Function x -> Not (x x))(Function x -> Not (x x))))}}}{\texttt{Not ((Function x -> Not (x x)) (Function x -> Not (x x)))}}}{\texttt{(Function x -> Not (x x)) (Function x -> Not (x x))}}$$

We know that `Not (Not (`e`))` evaluates to the same value as e. [5] We can see that we're going in circles. Again, this statement tells us that $P\ P \Rightarrow$ `True` if and only if $P\ P \Rightarrow$ `False`.

This is not how Russell viewed his paradox, but it has the same core structure; it is simply rephrased in terms of computation, and not set theory. The computational realization of the paradox is that the predicate doesn't compute to true or false, so its not a sensible logical statement. Russell's discovery of this paradox in Frege's set theory shook the foundations of mathematics. To solve this problem, Russell developed his ramified theory of types, which is the ancestor of types in programming languages. The program

```
(function x -> not(x x)) (function x -> not(x x))
```

is not typeable in OCaml for the same reason the corresponding predicate is not typeable in Russell's ramified theory of types. Try typing the above code into the OCaml top-level and see what happens.

More information on Russell's Paradox may be found in [14].

Encoding Recursion by Passing Self

In the logical view, passing a function to itself as argument is a bad thing. From a programming view, however, it can be an extremely powerful tool: passing a function to itself allows recursive functions to be defined without the use of `Let Rec`. We now show how we can modify the paradoxical combinator to do useful work. First, let's start with the original Mr. Bad:

```
(Function x -> Not (x x)) (Function x -> Not (x x))
```

The first step to making Mr. Bad do some good is rather than making an unbounded number of `Not`s, `Not (Not ( Not( ...)))`, lets make an unbounded number of some predefined function `F`:

```
(Function x -> F (x x)) (Function x -> F (x x))
```

Well, that makes unbounded `F (F ( F( ...)))`, but this sequence never terminates and so it doesn't do us any good. The next step is to *freeze* the computation at certain points to directly stop it from continuing; recall from our freeze macro above all we need to do is wrap some expression in a `Function _ -> ...` [6] to freeze it:

```
makeFroFs ≝
(Function x -> F (Function _ -> x x)) (Function x -> F (Function _ -> x x))
```

[5] In this case, anyway, but not in general. General assertions about the equivalence of expressions are hard to prove. In Section 2.4, we will explore a formal means of determining if two expressions are equivalent.

[6] We use "_" to represent a wildcard pattern, just like in OCaml.

Now, we have postponed the infinite execution; supposing `F` were say

```
Function froF -> True
```

the above would compute to

```
F (Function _ -> makeFroFs)
```

and since the `F` we defined throws away its argument this computation would in turn terminate with value `True`.

This particular example terminated *too* well, it threw away all the copies of `F` we painstakingly made. The way we can get recursion is to make an `F` which sometimes uses its argument `Function _ -> makeFroFs` to make recursive calls by *thawing* it. Consider the following revised `F`, aiming to ultimately be a function that sums the integers $\{0, 1, \ldots, n\}$ for argument n:

```
F ≝ Function froFs -> Function n ->
    If n = 0 Then 0 Else n + froFs 0 (n - 1)
```

If `makeFroFs` were to use this `F`, it would again compute to `F (Function _ -> makeFroFs)`, but the argument `Function _ -> makeFroFs` is not thrown out by this new `F`. Consider the computation of `makeFroFs 5`, by the above it is equivalent to computing

```
(F (Function _ -> makeFroFs)) 5
```

and so for the above definition of `F` we see parameter `froFs` will be instantiated with `Function _ -> makeFroFs`, and parameter `n` with `5`. Because `5 = 0` is `False` we then compute the else clause which after the above instantiation is

```
5 + (Function _ -> makeFroFs) 0 (5-1)
```

And, to compute the right-hand side of the addition, first the `0` dummy argument is applied to un-freeze `makeFroFs`, so we are now setting to compute `makeFroFs (5-1)`. Look above – this is nearly the exact spot we started at except the argument is one smaller! So, `makeFroFs` is now a recursive summation function and this example will ultimately compute to `15`. We have succeeded in repurposing the paradoxical combinator to write recursive functions.

There are a few minor clean-up steps we can perform on the above. First, since the `F` we care about are curried functions of two arguments (we need the additional argument e.g. `n` for the recursive call at a different value), we can make a revised freezer `Function n -> F n` which doesn't need to use `0` as the thawer but can "pun" and use the argument itself to do the thawing. So, we can redo the above definitions as

```
makeFs ≝ (Function x -> F' (Function n -> (x x) n))
(Function x -> F' (Function n -> (x x) n))
```

```
F' ≝ Function fs -> Function n ->
   If n = 0 Then 0 Else n + fs (n - 1)
```

– we can remove the `0` argument from the recursive call here since the `n-1` argument is doing the unfreezing work via our pun.

One last refactoring we can do to clean this up is to make a generic recursive-function-maker, by pulling out the `F'` in `makeFs` above as an explicit parameter `f`. This gives us

```
Y ≝ Function f ->
(Function x -> f (Function n -> (x x) n))
 (Function x -> f (Function n -> (x x) n))
```

and we can apply to some concrete `F`, e.g. `Y F'`, to create our recursive summing function. We call the above expression `Y` because this recursive function creator was discovered by logicians many years ago and given that name.

Another route to Y via direct self-passing

Since the construction of `Y` is complex we now present another path to its construction. This approach is more based on how object-oriented languages such as C++ implement messaging to self: every time an object method is invoked, (a pointer to) the object itself is passed as an additional parameter.

We demonstrate this approach by again writing a *summate* function. This time we follow the C++ idea and write the function to accept *two* arguments: `arg`, which is the number to which we will sum, and `this`, which is the copy of the function we require to be passed in so that recursion may occur. We will name our function `summate0` to reflect the fact that it is not quite the summate we want. It can be defined as

```
summate0 ≝ Function this -> Function arg ->
   If arg = 0 Then 0 Else arg + this this (arg - 1)
```

Note the use of `this this (arg - 1)`. The first use of `this` names the function to be applied; the second use of `this` is one of the arguments to that function. The argument `this` allows the recursive call to invoke the function again, thus allowing us to recurse as much as we need.

We can now sum the integers $\{0, 1, \ldots, 7\}$ with the expression

```
summate0 summate0 7
```

`summate0` always expects its first argument this to be itself. It can then use one copy for the recursive call (the first `this`) and pass the other copy on for future duplication. So `summate0 summate0` "primes the pump", so to speak, by giving the process an initial extra copy of itself.

Better yet, recall that currying allows us to obtain the inner function without applying it. In essence, a function with multiple arguments could be partially evaluated, with some arguments fixed and others waiting for input. We can use this to our advantage to define the summate function we want:

```
summate ≝ summate0 summate0
```

This allows us to hide the self-passing from the individual using our `summate` function, which cleans things up considerably. We can summarize the entire process as follows, recalling that even `Let` itself could be a macro for a function call:

```
summate ≝ Let summ = Function this -> Function arg ->
   If arg = 0 Then 0 Else arg + this this (arg - 1)
In summ summ
```

We now have a model for defining recursive functions without the use of the `Let Rec` operator. This means that untyped languages with no built-in recursion can still be Turing-complete. While this is an accomplishment, we can do even better; we can abstract the idea of self-passing almost entirely out of the body of the function itself.

The Y-Combinator The Y-combinator is a further abstraction of self-passing. The idea is that the Y-combinator does the self-application with an abstract body of code that is passed in as an argument. We first define a function called `almostY`, and then revise that definition to arrive at the real Y-combinator.

```
almostY ≝ Function body -> body body
```

using `almostY`, we can define `summate` as follows.

```
summate ≝ almostY (Function this -> Function arg ->
   If arg = 0 Then 0 Else arg + this this (arg - 1))
```

The true Y-combinator actually goes one step further and allows us to write recursive calls in the more natural style of just "`this (arg - 1)`", avoiding the extra `this` parameter. To do this, we assume that the `body` argument is already in this simple form. We then define a new form, `wrapper`, which replaces `this` with `(this this)` in `body`:

Definition 2.13 (Y-Combinator)**.**

```
combY =def Function body ->
    Let wrapper = Function this -> Function arg -> body (this this) arg
    In wrapper wrapper
```

This is the same `Y` as was defined in the previous subsection.

The transformation steps performed in these examples are also good examples of the power of higher-order functional programming: "code surgery" is performed on `body` to produce `wrapper` by simply using function abstraction and application. The Y-combinator can then be used to define `summate` as

```
summate =def combY (Function this -> Function arg ->
    If arg = 0 Then 0 Else arg + this (arg - 1))
```

2.3.6 Call-By-Name Parameter Passing

In **call-by-name** parameter passing, the argument to the function is not evaluated at function call time, but rather is only evaluated if it is used. This style of parameter passing is largely of historical interest now; Algol uses it but no modern languages are call-by-name by default (The Digital Mars D language does allow parameters to be treated as call-by-name via use of the `lazy` qualifier). The reason is that it is much harder to write efficient compilers if call-by-name parameter passing is used. Nonetheless, it is worth taking a brief look at call-by-name parameter passing.

Let us define the operational semantics for call-by-name.

(Call-By-Name Application)
$$\frac{e_1 \Rightarrow \texttt{Function}\ x\ \texttt{->}\ e, \quad e[e_2/x] \Rightarrow v}{e_1\ e_2 \Rightarrow v}$$

Freezing and thawing, defined in Section 2.3.4, is a way to get call-by-name behavior in a call-by-value language. Consider, then, the computation of

```
(Function x -> Thaw x + Thaw x) (Freeze (3 - 2))
```

`(3 - 2)` is not evaluated until we are inside the body of the function where it is thawed, and it is then evaluated two separate times. This is precisely the behavior of call-by-name parameter passing, so `Freeze` and `Thaw` can encode it by this means. The fact that `(3 - 2)` is executed twice shows the main weakness of call by name, namely repeated evaluation of the function argument.

Lazy or **call-by-need** evaluation is a version of call-by-name that caches evaluated function arguments the first time they are evaluated so it doesn't have to re-evaluate them in subsequent uses. Haskell [13, 7] is a pure functional language with lazy evaluation.

2.3.7 F♭ Abstract Syntax

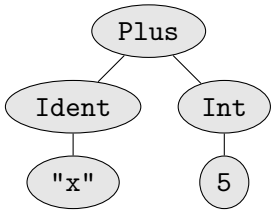
The previous sections thoroughly describe the operational semantics of **F♭** in terms of its concrete syntax. Recall from our boolean language, however, that an interpreter operates over a representation of a program which is more conducive to programmatic manipulation; this representation is termed its **abstract syntax**. To define the abstract syntax of **F♭** for a OCaml interpreter, we need to define a variant type that captures the expressiveness of **F♭**. The variant types we will use are as follows.

```
type ident = Ident of string

type expr =
 Var of ident | Function of ident * expr | Appl of expr * expr |
 Let of ident * expr * expr | LetRec of ident * ident * expr * expr |
 Plus of expr * expr | Minus of expr * expr | Equal of expr * expr |
 And of expr * expr| Or of expr * expr | Not of expr |
 If of expr * expr * expr | Int of int | Bool of bool

type fbtype = TInt | TBool | TArrow of fbtype * fbtype | TVar of string;;
```

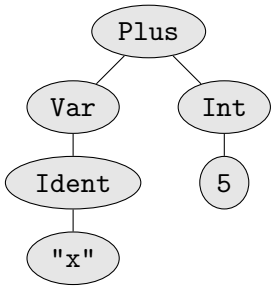
One important point here is the existence of the `ident` type. Notice where `ident` is used in the `expr` type: as variable identifiers, and as function parameters for `Function` and `Let Rec`. The `ident` type attaches additional semantic information to a string, indicating that the string specifically represents an identifier.

Note, though, that `ident` is used in two different ways: to signify the *declaration* of a variable (such as in `Function (Ident "x",...)`) and to signify the *use* of a variable (such as in `Var(Ident "x")`). Observe that the use of a variable is an expression and so can appear in the AST anywhere that any expression can appear. The declaration of a variable, on the other hand, is not an expression; variables are only declared in functions. In this way, we are able to use the OCaml type system to help us keep the properties of the AST nodes straight.

For example, consider the following AST:



At first glance, it might appear that this AST represents the **F♭** expression `x + 5`. However, the AST above cannot actually exist using the variants we defined above. The `Plus` variation accepts two expressions upon construction and `Ident` is not a variant; thus, the equivalent OCaml code `Plus(Ident "x",Int 5)` would not even typecheck. The **F♭** expression `x + 5` is represented instead by the AST



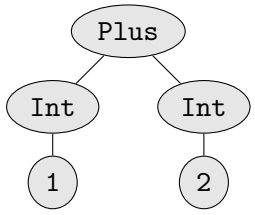
which is represented by the OCaml code `Plus(Var(Ident "x"),Int 5)`.

Being able to convert from abstract to concrete syntax and vice versa is an important skill for one to develop, but it takes some time to become proficient at this conversion. Let us look as some examples **F**♭ in pursuit of refining this skill.

Example 2.17.

Concrete:
```
1 + 2
```

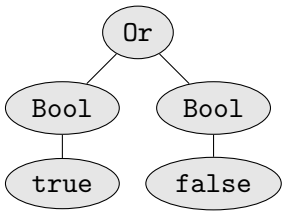
Abstract:
```
Plus(Int 1,Int 2)
```



Example 2.18.

Concrete:
```
True Or False
```

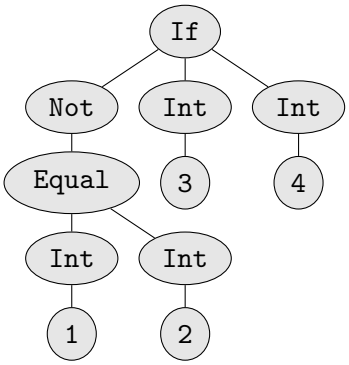
Abstract:
```
Or(Bool true,Bool false)
```



Example 2.19.

Concrete:
```
If Not(1 = 2) Then 3 Else 4
```

Abstract:
```
If(Not(Equal(Int 1, Int 2)),
   Int 3, Int 4)
```



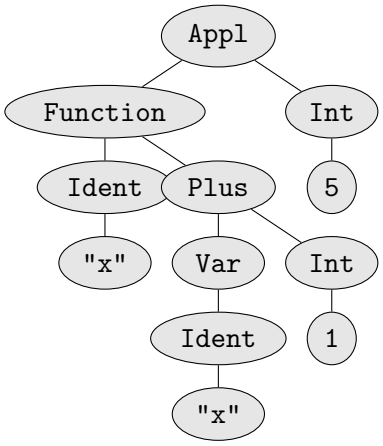
Example 2.20.

Concrete:

```
(Function x -> x + 1) 5
```

Abstract:

```
Appl(
  Function(
    Ident "x",
    Plus(Var(Ident "x"),
      Int 1)),
  Int 5)
```



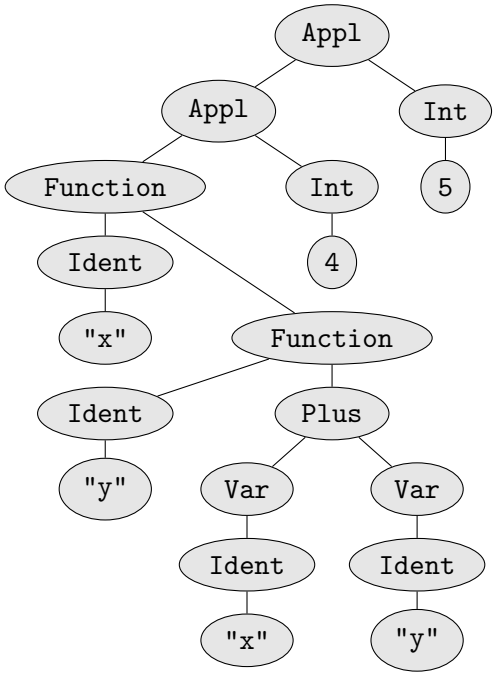
Example 2.21.

Concrete:

```
(Function x -> Function y ->
  x + y) 4 5
```

Abstract:

```
Appl(
  Appl(
    Function(
      Ident "x",
      Function(
        Ident "y",
        Plus(
          Var(Ident "x"),
          Var(Ident "y")
    ))),
    Int 4),
  Int 5)
```



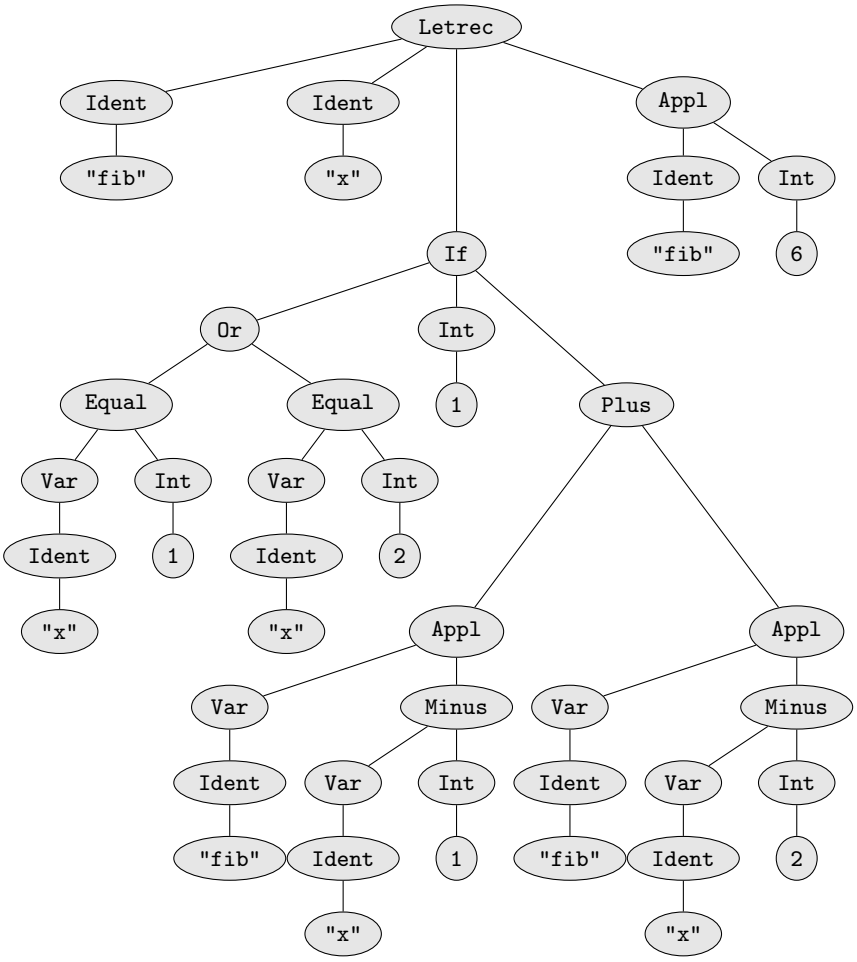
Example 2.22.

Concrete:

```
Let Rec fib x =
   If x = 1 Or x = 2 Then 1 Else fib (x - 1) + fib (x - 2)
In fib 6
```

Abstract:

```
Letrec(Ident "fib", Ident "x", If(Or(Equal(Var(Ident "x"), Int 1),
Equal(Var(Ident "x"), Int 2)), Int 1, Plus(Appl(Var(Ident "fib"),
Minus(Var(Ident "x"), Int 1)), Appl(Var(Ident "fib"), Minus(Var(Ident
"x"), Int 2)))), Appl(Var(Ident "fib"), Int 6))
```



Notice how lengthy even simple expressions can become when represented in the abstract syntax. Review the above examples carefully, and try some additional examples of your own. It is important to be able to comfortably switch between abstract and concrete syntax when writing compilers and interpreters.

2.4 Operational Equivalence

One of the most basic operations defined over a space of mathematical objects is the equivalence relation ($\cong$). We have, using operational semantics, defined programs as mathematical entities. As a result, equivalence makes sense for programs too.

First, we are compelled to consider what we mean intuitively by "equivalent" programs. Two things which are equivalent are interchangable; thus, two equivalent programs must do the same thing. It is not necessary that they do so in exactly the same way however. For example, we can easily see that the programs `(1 + 1 + 1 - 1)` and `(2)` are equivalent, but the prior requires more computation to evaluate.

Because two equivalent programs can be substituted for each other but may have different execution-time properties, we can use operational equivalence when optimizing compilers and other such tools. Operational equivalence provides a rigorous and reliable foundation for such work; we do not need to worry about an optimization changing the behavior of an application because we can prove mathematically that such a change is impossible. Operational equivalence also defines the process of refactoring, a process by which developers can change the structure of a program for maintainability or enhancement without changing the program's behavior.

Defining an equivalence relation for programs is actually not as straightforward as one might expect. The initial idea is to define the relation such that two programs are equivalent if they always lead to the same results when used. As we will see, however, this definition is not sufficient, and we will need to do some work to arrive at a satisfactory definition.

Let us begin by looking at a few sample equivalences to get a feel for what they are. η-conversion (or *eta*-conversion) is one example of an interesting equivalence. It is defined as follows.

`Function` x `->` $e \cong$
`Function` z `->` `(Function` x `->` e`)` z, for z not free in e

η-conversion is similar to the proxy pattern in object oriented programming[12]. A closely related law for our freeze/thaw syntax is

`Thaw (Freeze` e`)` $\cong e$

In both examples, one of the expressions may be replaced by the other without ill effects (besides perhaps changing execution time), so we say they are equivalent. To write formal proofs, however, we will need to develop a more rigorous definition of equivalence.

2.4.1 Defining Operational Equivalence

Let's begin by informally strengthening our definition of operational equivalence. We define equivalence in a manner dating all the way back to Leibniz[18]:

Definition 2.14 (Operational Equivalence (Informal))**.** *Two program expressions are* equivalent *if and only if one can be replaced with the other at any place, and no external change in behavior will be noticed.*

We wish to study equivalence for possibly open programs, because there are good equivalences such as x `+ 1 - 1` $\cong x+0$. We define "at any place" by the notion of a **program context**, which is, informally, a **F**♭ program with some holes (•) in it. Using this informal definition, testing if $e_1 \cong e_2$ would be roughly equivalent to performing the following steps (for all possible programs and all possible holes, of course).

1. Select any program context (that is, program containing a hole).
2. Place e_1 in the • position and run the program.
3. Do the same for e_2.
4. If the observable result is different, e_1 is not equivalent to e_2.
5. Repeat steps 1-4 for *every possible context*. If none of these infinitely many contexts produces different results, then e_1 is equivalent to e_2.

Now let us elaborate on the notion of a program context. Take an **F**♭ program with some "holes" (•) punched in it: replace some subterms of any expression with •. Then "hole-filling" in this program context C, written $C[e]$, means replacing • with e in C. Hole filling is like substitution, but without the concerns of bound or free variables. It is direct replacement with no conditions.

Let us look at an example of contexts and hole-filling using η-conversion as we defined above. Let

$C \stackrel{\text{def}}{=}$ `(Function z -> Function x -> •) z`

Now, filling the hole with x `+ 2` is simply written

```
((Function z -> Function x -> •) z)[x + 2] =
      (Function z -> Function x -> x + 2) z
```

Finally, we are ready to rigorously define operational equivalence.

Definition 2.15 (Operational Equivalence)**.** $e \cong e'$ *if and only if for all contexts* C *such that* $C[e]$ *and* $C[e']$ *are both closed expressions,* $C[e] \Rightarrow v$ *for some* v *if and only if* $C[e'] \Rightarrow v'$ *for some* v'*.*

Another way to phrase this definition is that two expressions are equivalent if in any possible context, C, one terminates if the other does. We call this *operational* equivalence because it is based on the interpreter for the language, or rather it is based on the operational semantics. The most interesting, and perhaps nonintuitive, part of

this definition is that nothing is said about the relationship between v and v'. In fact, they may be different in theory. However, intuition tells us that v and v' must be very similar, since equivalence holds for any possible context.

The only problem with this definition of equivalence is its "incestuous" nature—there is no absolute standard of equivalence removed from the language. **Domain theory** is a mathematical discipline which defines an algebra of programs in terms of existing mathematical objects (complete and continuous partial orders). We are not going to discuss domain theory here, mainly because it does not generalize well to programming languages with side effects. [16] explores the relationship between operational semantics and domain theory.

2.4.2 Properties of Operational Equivalence

In this section, we present some general equivalence principles for $\mathbf{F}\flat$.

Definition 2.16 (Reflexivity)**.**

$e \cong e$

Definition 2.17 (Symmetry)**.**

$e \cong e'$ *if* $e' \cong e$

Definition 2.18 (Transitivity)**.**

$e \cong e''$ *if* $e \cong e'$ *and* $e' \cong e''$

Definition 2.19 (Congruence)**.**

$C[e] \cong C[e']$ *if* $e \cong e'$

Definition 2.20 (β-Equivalence)**.**

$(\texttt{(Function}\ x\ \texttt{->}\ e\texttt{)}\ v) \cong (e[v/x])$

provided v is closed (if v had free variables they could be captured when v is placed deep inside e).

Definition 2.21 (η-Equivalence)**.**

$(\texttt{Function}\ x\ \texttt{->}\ e) \cong (\texttt{Function}\ z\ \texttt{->}\ (\texttt{Function}\ x\ \texttt{->}\ e)\ z)$

Definition 2.22 (α-Equivalence)**.**

$(\texttt{Function}\ x\ \texttt{->}\ e) \cong (\texttt{Function}\ y\ \texttt{->}\ (e[y/x]))$

Definition 2.23.

$(n\ \texttt{+}\ n') \cong$ *the sum of* n *and* n'

Similar rules hold for `-`, `And`, `Or`, `Not`, *and* `=`.

Definition 2.24.

$(\texttt{If True Then } e \texttt{ Else } e') \cong e$

A similar rule holds for `If False...`

Definition 2.25.

If $e \Rightarrow v$ *then* $e \cong v$

Equivalence transformations on programs can be used to justify results of computations instead of directly computing with the interpreter; it is often easier. An important component of compiler optimization is applying transformations, such as the ones above, that preserve equivalence.

2.4.3 Examples of Operational Equivalence

To solidify the concept of operational equivalence (one which reliably boggles newcomers to programming language theory), we provide a number of examples of equivalent and non-equivalent expressions. We start with a very simple example of two expressions which are not equivalent.

Lemma 2.5. *`2` $\ncong$ `3`*

Proof. By example. Let $C \stackrel{\text{def}}{=}$ `If •= 2 Then 0 Else (0 0)`. $C[\texttt{2}] \Rightarrow \texttt{0}$ while $C[\texttt{3}] \not\Rightarrow v$ for any v. Thus, by definition, `2` $\ncong$ `3`. □

Note that, in the above proof, we used the expression `(0 0)`. This expression cannot evaluate; the application rule applies only to functions. As a result, this expression makes an excellent tool for intentionally making code get stuck when building a proof about operational equivalence. Other similar get-stuck expressions exist, such as `True + True`, `Not 5`, and the ever-popular `(Function x -> x x)(Function x -> x x)`.

It should be clear that we can use the approach in Lemma 2.5 to prove that any two integers which are not equal are not operationally equivalent. But can we make a statement about a non-value expression?

Lemma 2.6. *`x` $\ncong$ `x + 1 - 1`.*

At first glance, this inequivalence may seem counterintuitive. But the proof is fairly simple:

Proof. By example. Let $C \stackrel{\text{def}}{=}$ `(Function x -> •) True`. Then $C[\texttt{x}] \Rightarrow$ `True`. $C[\texttt{x + 1 - 1}] \equiv$ `(Function x -> x + 1 - 1) True`, which cannot evaluate because no rule allows us to evaluate `True + 1`. Thus, by definition, `x` $\ncong$ `x + 1 - 1`. □

Similar proofs could be used to prove inequivalences such as `Not(Not(`e`))` $\ncong e$. The key here is that some of the rules in **F♭** distinguish between integer values and boolean values. As a result, equivalences cannot hold if one side makes assumptions about the type of value to which it will evaluate and the other side does not.

We have proven inequivalences; can we prove an equivalence? It turns out that some equivalences can be proven but that this process is much more involved. Thus far, our proofs of inequivalence have relied on the fact that they merely need to demonstrate an example context in which the expressions produce different results. A proof of equivalence, however, must demonstrate that no such example exists among infinitely many contexts. For example, consider the following:

Lemma 2.7. *If $e \not\Rightarrow v$ for any v, $e' \not\Rightarrow v'$ for any v', and both e and e' are closed expressions, then $e \cong e'$. For example,* `(0 0)` $\cong$ `(Function x -> x x)(Function x -> x x)`.

First, we need a definition:

Definition 2.26 (Touch)**.** *An expression is said to* ***touch*** *one of its subexpressions if the evaluation of the expression causes the evaluation of that subexpression.*

That is, `If True Then 0 Else 1` *touches* the subexpression `0` because it is evaluated when the whole expression is evaluated. `1` is not touched because the evaluation of the expression never causes `1` to be evaluated.

We can now prove Lemma 2.7.

Proof. By contradiction. Without loss of generalization, suppose that there is some context C for which $C[e] \Rightarrow v$ while $C[e'] \not\Rightarrow v'$ for any v'.

Because e is a closed expression, $C'[e] \not\Rightarrow v$ for all v and all contexts C' which touch e. Because $C[e] \Rightarrow v$, we know that C does not touch the hole.

Because C does not touch the hole, how the hole evaluates cannot affect the evaluation of C; that is, $C[e^*] \Rightarrow v^*$ for some v^* must be true for all e^* or for no e^*.

Because $C[e] \Rightarrow v$, $C[e^*] \Rightarrow v^*$ for some v^* for all e^*. But $C[e'] \not\Rightarrow v'$ for any v'. Thus, by contradiction, C does not exist. Thus, by definition, $e \cong e'$. □

The above proof is much longer and more complex than any of the proofs of inequivalence that we have encountered and it isn't even very robust. It could benefit, for example, from a proof that a closed expression cannot be changed by a replacement rule (which is taken for granted above). Furthermore, the above proof doesn't even prove a very useful fact! Of far more interest would be proving the operational equivalence of two expressions when one of them is executed in a more desirable manner than the other (faster, fewer resources, etc.) in order to motivate compiler optimizations.

Lemma 2.7 is, however, effective in demonstrating the complexities involved in proving operational equivalence. It is surprisingly difficult to prove that an equivalence holds; even proving `1 + 1` $\cong$ `2` is quite challenging. See [16] for more information on this topic.

2.4.4 The λ-Calculus

We briefly consider the **λ-calculus**. In Section 2.3, we saw how to encode tuples, lists, `Let` statements, freezing and thawing, and even recursion in **F♭**. The encoding approach is very powerful, and also gives us a way to understand complex languages based on our understanding of simpler ones. Even numbers, booleans, and if-then-else statements are encodable, although we will skip these topics. Thus, all that is needed is functions

and application to make a Turing-complete programming language. This language is known as the **pure λ calculus**, because functions are usually written as $\lambda x.e$ instead of `Function` x `->` e.

Execution in λ calculus is extremely straightforward and concise. The main points are as follows.

- Even programs with free variables can execute (or *reduce* in λ-calculus terminology).
- Reduction can happen anywhere, e.g. inside a function body that hasn't been called yet.
- $(\lambda x.e)e' \Rightarrow e[e'/x]$ is the only reduction rule, called β-reduction. (It has a special side-condition that it must be *capture-free*, i.e. no free variables in e' become bound in the result. Capture is one of the complications of allowing reduction anywhere.)

This form of computation is conceptually very interesting, but is more distant from how actual computer languages execute and so we do not put a strong focus on it here.

Exercises

Exercise 2.1. How would you change the Sheep language to allow the terms $bah, baah, \cdots$ without excluding any terms which are already allowed?

Exercise 2.2. Is the term it in the Frog language? Why or why not?

Exercise 2.3. Is it possible to construct a term in Frog without using the terminal t? If so, give an example. If not, why not?

Exercise 2.4. Complete the definition of the operational semantics for the boolean language described in section 2.2.1 by writing the rules for `Or` and `Implies`.

Exercise 2.5. Why not just use interpreters and forget about the operational semantics approach?

Chapter 3

Tuples, Records, and Variants

In Chapter 2 we saw that, using a language with only functions and application, we could represent advanced programming constructs such as tuples and lists. However, we pointed out that these encodings have fundamental problems, such as a low degree of efficiency, and the fact that they necessarily expose their details to the programmer, making them difficult and dangerous to work with in practice. Recall how we could take our encoding of a pair from Chapter 2 and apply it like a function; clearly the wrong behavior. In this chapter we look at how we can build some of these advanced features into the language, namely tuples and records, and we conclude the chapter by examining variants.

3.1 Tuples

One of the most fundamental forms of data aggregation in programming is the notion of **pairing**. With pairs, or 2-tuples, almost any data structure can be represented. Tripling can be represented as $(1, (2, 3))$, and in general n-tuples can be represented with pairs in a similar fashion. Records and C-style structs can be represented with sets (n-tuples) of (label, value)-pairs. Even objects can be built up from pairs, but this is stretching it (just as encoding pairs as functions was stretching it).

In Chapter 2, we showed an encoding of pairs based on functions. There were two problems with this representation of pairs. First of all, the representation was inefficient. More importantly, the behavior of pairs was slightly wrong, because we could apply them like functions. To really handle pairs correctly, we need to add them directly to the language. We can add pairs to $\mathbf{F}\flat$ in a fairly straightforward manner. We show how to add pair functionality to the interpreter, and leave the operational semantics for pairs as an exercise for the reader.

First, we extend the `expr` type in our interpreter to include the following.

```
type expr =
  ...
| Pr of expr * expr | Left of expr | Right of expr
```

Next, we add the following clauses to our `eval` function.

```
let rec eval e =
  match e with
    ...
  | Pr(e1, e2) -> Pr(eval e1, eval e2)
  | Left(expr) -> (match eval expr with
      Pr(e1,e2) -> e1
    | _ -> raise TypeMismatch)
  | Right(expr) -> (match eval expr with
      Pr(e1, e2) -> e2
    | _ -> raise TypeMismatch)
```

Notice that our pairs are *eager*, that is, the left and right components of the pair are evaluated, and must be values for the pair itself to be considered a value. For example, `(2, 3+4)` $\Rightarrow$ `(2, 7)`. OCaml tuples exhibit this same behavior. Also notice that our space of values is now bigger. It includes:

- numbers `0`, `1`, `-1`, `2`, `-2`, ...
- booleans `True`, `False`
- functions `Function x ->` ...
- pairs (v_1, v_2)

Exercise 3.1. How would we write our interpreter to handle pairs in a non-eager way? In other words, what would need to be in the interpreter so that (e_1, e_2) was considered a value (as opposed to only (v_1, v_2) being considered a value)?

Now that we have 2-tuples, encoding 3-tuples, 4-tuples, and n-tuples is easy. We simply do them as $(1, (2, (3, (\ldots, n))))$. As we saw before, lists can be encoded as n-tuples.

3.2 Records

Records are a variation on tuples in which the fields have names. Records have several advantages over tuples. The main advantage is the named field. From a software engineering perspective, a named field "zipcode" is far superior to "the third element in the tuple." The order of the fields of a record is arbitrary, unlike with tuples.

Records are also far closer to objects than tuples are. We can encode object polymorphism via record polymorphism. Record polymorphism is discussed in Section 3.2.1. The motivation for using records to encode objects is that a subclass is composed of a superset of the fields of its parent class, and yet instances of both classes may be used in the context of the superclass. Similarly, record polymorphism allows records to be used in the context of a subset of their fields, and so the mapping is quite natural. We will use records to model objects in Chapter 5.

Our **F♭** records will have the same syntax as OCaml records. That is, records are written as $\{l_1{=}e_1\,;\ l_2{=}e_2\,;\ \ldots\,;\ l_n{=}e_n\}$, and selection is written as $e.l_k$, which selects

the value labeled l_k from record e. We use l as a metavariable ranging over labels, just as we use e as a metavariable indicating an expression; an actual record is for instance {`x=5; y=7; z=6`}, so `x` here is an actual label.

If records are always statically known to be of fixed size, that is, if they are known to be of fixed size at the time we write our code, then we may simply map the labels to integers, and encode the record as a tuple. For instance,

{`x=5; y=7; z=6`} = `(5, (7, 6))`
e`.x` = `Left` e
e`.y` = `Left (Right` e`)`
e`.z` = `Right (Right` e`)`

Obviously, the makes for ugly, hard-to-read code, but for C-style structs, it works. But in the case where records can shrink and grow, this encoding is fundamentally too weak. C++ structs can be subtypes of one another, so fields that are not declared may, in fact, be present at runtime.

On the other hand, pairs can be encoded as records quite nicely. The pair `(3, 4)` can simply be encoded as the record {`l=3; r=4`}. More complex pairs, such as those used to represent lists, can also be encoded as records. For example, the pair `(3, (4, (5, 6)))`, which represents the list `[3; 4; 5; 6]`, can be encoded as the record {`l=3; r=`{`l=4; r=`{`l=5; r=`{`l=6; r=emptylist`}}}}.

A variation of this list encoding is used in the mergesort example in Section 4.3.2. This variation encodes the above list as {`l=3; r=`{`l=4; r=`{`l=5; r=`{`l=6; r=emptylist`}}}}. This encoding has the nice property that the values are always contained in the `l` fields, and the rest of the list is always contained in the `r` fields. This is much closer to the way real languages such as OCaml, Scheme, and Lisp represent lists (recall how we write statements like `let (first::rest) = mylist` in OCaml).

3.2.1 Record Polymorphism

Records do more that just add readability to programs. For instance, if you have {`size=10; weight=100`} and {`weight=10; name="Mike"`}, either of these two records can be passed to a function such as

```
Function x -> x.weight.
```

In scripting languages, this flexibility is informally known as *duck typing* – if `Function d -> d.quack ()` does not produce a run-time error when invoked we will assume the `d` passed to it is a duck since it quacked like a duck.

This form of flexibility is called **subtype polymorphism** in a typed language. In the function above, `x` can be any record with a `weight` field. Subtype polymorphism on records is known as **record polymorphism**. OCaml disallows record polymorphism, so the OCaml version of the above code will not typecheck.

In typed object-oriented languages, subtype polymorphism is known as **object polymorphism**, or, more commonly, as simply *polymorphism*. The latter is, unfortunately, confusing with respect to the parametric polymorphism of OCaml.

3.2.2 The F♭R Language

We will now define the **F♭R** language: **F♭** with records. Again, we will concentrate on the interpreter, and leave the operational semantics as an exercise to the reader.

The first thing we need to consider is how to represent record labels. Record labels are symbols, and so we could use our identifiers (`Ident "x"`) as labels, but it is better to think of record labels as a different sort. For instance, labels are never bound or substituted for. So we will define a new type in our interpreter.

```
type label = Lab of string
```

Next, we need a way to represent the record itself. Records may be of arbitrary length, so a list of (*label*, *expression*)-pairs is needed. In addition, we need a way to represent selection. The **F♭R** `expr` type now looks like the following.

```
type expr = ...
| Record of (label * expr) list | Select of expr * label
```

Let's look at some concrete to abstract syntax examples for our new language.

Example 3.1.

```
{size=7; weight=255}

Record [(Lab "size", Int 7); (Lab "weight", Int 255)]
```

Example 3.2.

```
e.size

Select(Var(Ident "e"), Lab "size")
```

In addition, our definition of values must now be extended to include records. Specifically, $\{l_1{=}v_1\,;\ l_2{=}v_2\,;\ \ldots\,;\ l_n{=}v_n\}$ is a value, provided that $v_1, v_2, \ldots, v_n$ are values.

Finally, we add the necessary rules to our interpreter. Because records can be of arbitrary length, we will have to do a little more work when evaluating them. The `let-rec-and` syntax in OCaml is used to declare mutually recursive functions, and we use it below.

```
(* A function to project a given field *)

let rec lookupRecord body (Lab l) = match body with
  [] -> raise FieldNotFound
| (Lab l', v)::t -> if l = l' then v else lookupRecord t (Lab l)
```

```
(* The eval function, with an evalRecord helper *)

let rec eval e = match e with
  ...
| Record(body) -> Record(evalRecord body)
| Select(e, l) -> match eval e with
    Record(body) -> lookupRecord body l
  | _ -> raise TypeMismatch

and evalRecord body = match body with
  [] -> []
| (Lab l, e)::t -> (Lab l, eval e)::evalRecord t
```

Notice that our interpreter correctly handles {}, the empty record, by having it compute to the itself since it is, by definition, a value.

Interact with F♭SR. We can use our **F♭SR** interpreter to explore records (**F♭SR** is **F♭** with records and state, and is introduced in Chapter 4). First, let's try a simple example to demonstrate the eager evaluation of records.

```
# {one = 1; two = 2;
   three = 2 + 1; four = (Function x -> x + x) 2};;
==> {one=1; two=2; three=3; four=4}
```

Next, let's try a more interesting example, where we use records to encode lists. Note that we define `emptylist` as `-1`. The function below sums all values in a list (assuming it has a list of integers).

```
# Let emptylist = 0 - 1 In
  Let Rec sumlist list =
    If list = emptylist Then
      0
    Else
      (list.l) + sumlist (list.r) In
  sumlist {l=1; r={l=2; r={l=3; r={l=4; r=emptylist}}}};;
==> 10
```

3.3 Variants

We have been using variants in OCaml, as the types for expressions `expr`. Now we study untyped variants more closely. OCaml actually has two (incompatible) forms of variant, regular variants and polymorphic variants . In the untyped context we are working in, the OCaml polymorphic variants are more appropriate and we will use that form.

We briefly contrast the two forms of variant in OCaml for readers unfamiliar with polymorphic variants. Recall that in OCaml, regular variants are first declared as types

```
type feeling =
    Vaguely of feeling | Mixed of feeling * feeling |
    Love of string | Hate of string | Happy | Depressed
```

which allows `Vaguely(Happy)`, `Mixed(Vaguely(Happy),Hate("Fred"))`, etc. Polymorphic variants require no type declaration; thus, for the above we can directly write `‘Vaguely(‘Happy)`, `‘Mixed(‘Vaguely(‘Happy),‘Hate("Fred"))`, etc. The ‘ must be prefixed each variant name, indicating it is a polymorphic variant name.

3.3.1 Variant Polymorphism

Like records, variants are polymorphic. In records, many different forms of record could get through a particular selection (any record with the selected field). In variants, the polymorphism is dual in that many different forms of `match` statement can process a given variant.

3.3.2 The F♭V Language

We will now define the **F♭V** language, **F♭** with . . . **V**ariants.

The new syntax requires variant syntax and match syntax. Just as we restrict functions to have one argument only, we also restrict variant constructors to take one argument only; multiple- or zero-argument variants must be encoded. In concrete syntax, we construct variants by $n(e)$ for n a named variant and e its parameter, for example `‘Positive(3)`. Variants are then used via match: `Match` e `With` n_1`(`x_1`) ->` e_1 `|` `...|` n_m`(`x_m`) ->` e_m. We don't define a general pattern `match` as found in OCaml—our `Match` will matching a single variant field at a time, and won't work on anything besides variants.

The abstract syntax for **F♭V** is as follows. First, each variant needs a name.

```
type name = Name of string
```

The **F♭V** abstract syntax `expr` type now looks like

```
type expr = ...
  | Variant of (name * expr)
  | Match of expr * (name * ident * expr) list
```

Let's look at some concrete to abstract syntax examples for our new language.

Example 3.3.

```
‘Positive(4)



Variant(Name "Positive", Int 4)
```

Example 3.4.

```
Match e With
   ‘Positive(x) -> 1 | ‘Negative(y) -> -1 | ‘Zero(p) -> 0


Match(Var(Ident("e")),[(Name "Positive",Ident "x",Int 1);
                       (Name "Negative",Ident "y",Int -1);
                       (Name "Zero",Ident "p",Int 0)])
```

Note in this example we can't just have a variant `Zero` since 0-ary variants are not allowed, and a dummy argument must be supplied. Multiple-argument variants may be encoded by a single argument variant over a pair or record (since we have neither pair or records in **F♭V**, the only recourse is the encoding of pairs used in **F♭** in Section 2.3.4).

In addition, our definition of **F♭V** values must also be extended from the **F♭** ones to include variants: $n(v)$ is a value, provided v is. To define the meaning of **F♭V** execution, we extend the operational semantics of **F♭** with the following two rules:

(Variant Rule)
$$\frac{e \Rightarrow v}{n(e) \Rightarrow n(v)}$$

(`Match` Rule)
$$\frac{e \Rightarrow n_j(v_j), \quad e_j[v_j/x_j] \Rightarrow v}{\begin{array}{l}\texttt{Match } e \texttt{ With} \\ \quad n_1(x_1) \texttt{ -> } e_1 \texttt{ | } \ldots \\ \texttt{ | } n_j(x_j) \texttt{ -> } e_j \texttt{ | } \ldots \\ \texttt{ | } n_m(x_m) \texttt{ -> } e_m\end{array} \Rightarrow \quad v}$$

The Variant rule constructs a new variant labeled n; its argument is eagerly evaluated to a value, just as in OCaml: `‘Positive(3+2)` $\Rightarrow$ `‘Positive(5)`. The `Match` rule first computes the expression e being matched to a variant $n_j(v_j)$, and then looks up that variant in the match, finding $n_j(x_j)$ `->` e_j, and then evaluating e_j with its variable x_j given the value of the variant argument.

Example 3.5.

```
Match ‘Grilled(3+1) With
  ‘Stewed(x) -> 4 + x |
  ‘Grilled(y) -> 2 + y    ⇒ 6, because
‘Grilled(3+1) ⇒ ‘Grilled(4) and (2 + y)[4/y] ⇒ 6
```

Exercise 3.2. Extend the **F♭V** syntax and operational semantics so the `Match` expression always has a final match of the form of the form"`| _ -> e`". Is this `Match` strictly more expressive than the old one, or not?

Variants and Records are Duals Here we see how the definition of a record is modeled as a use of a variant, and a use of a record is the definition of a variant.

Variants are the dual of records: a record is this field *and* that field *and* that field; a variant is this field *or* that field *or* that field. Since they are duals, *defining* a record looks something like *using* a variant, and defining a variant looks like using a record.

Variants can directly encode records and vice-versa, in a programming analogy of how DeMorgan's Laws allows logical and to be encoded in terms of or, and vice-versa: p `Or` $q = $ `Not(Not` p `And Not` q`)`; p `And` $q = $ `Not(Not` p `Or Not` q`)`.

Variants can be encoded using records as follows.

`Match` s `With` ‘$n_1(x_1)$ `->` e_1 `|` $\ldots$ `|` ‘$n_m(x_m)$ `->` $e_m =$
$\quad s\{n_1$`=Function` x_1 `->` e_1`;` $\ldots$`;` n_m`=Function` x_m `->` $e_m\}$
‘$n(e) =$ `(Function` x `->` `(Function` r `->` $r.n\ x$ `))` e

The tricky part of the encoding is that definitions must be turned in to uses and vice-versa. This is done with functions: an injection is modeled as a function which is *given* a record and will select the specified field.

Here is how records can be encoded using variants.

$\{l_1$`=`e_1`;` $\ldots$`;` l_n`=`$e_n\} =$ `(Function` k_1 `->`$\ldots$`Function` k_n `->Function` s `->`
$\quad$ `Match` s `With` ‘l_1`(`x`)` `->` k_1 `|` $\ldots$ `|` ‘l_n`(`x`)` `->` k_n`)(`e_1`)`$\ldots$`(`e_n`)`
$e.l_k = e$ ‘l_k`(0)`

where x above is any fresh variable.

One other interesting aspect about the duality between records and variants is that *both* records and variants can encode objects. A variant is a message, and an object is a case on the message. In the variant encoding of objects, it is easy to pass around messages as first-class entities. Using variants to encode objects makes objects that are hard to typecheck, however, and that is why we think of objects as more record-like.

Chapter 4

Side Effects: State and Exceptions

We will now leave the world of pure functional programming, and begin considering languages with side-effects. For now we will focus solely on two particular side-effects, state and exceptions. There are, however, many other types of side-effects, including the following.

- Goto-statements or loop-breaks, which are similar to exceptions. Note that loop-breaks require loops, which require state.
- Input and output.
- Distributed message passing.

4.1 State

Languages like **F♭**, **F♭R**, and **F♭V** are pure functional languages. Once we add any kind of side-effect to a language it is not pure functional anymore. Side-effects are non-local, meaning they can affect other parts of the program. As an example, consider the following OCaml code.

```
let x = ref 9 in
  let f z = x := !x + z in
    x := 5; f 5; !x
```

This expression evaluates to `10`. Even though `x` was defined as a reference to `9` when `f` was declared, the last line sets `x` to be a reference to `5`, and during the application of `f`, `x` is reassigned to be a reference to `(5 + 5)`, making it `10` when it is finally dereferenced in the last line. Clearly, the use of side effects makes a program much more difficult to analyze, since it is not as declarative as a functional program. When looking at programs with side-effects, one must examine the *entire* body of code to see which side-effects influence the outcome of which expressions. Therefore, it is a good programming moral to use side-effects only when they are strongly needed.

Let us begin by informally discussing the semantics of references. In essence, when a reference is created in a program, a cell is created that contains the specified value, and the reference itself is a pointer to that cell. A good metaphor for these reference

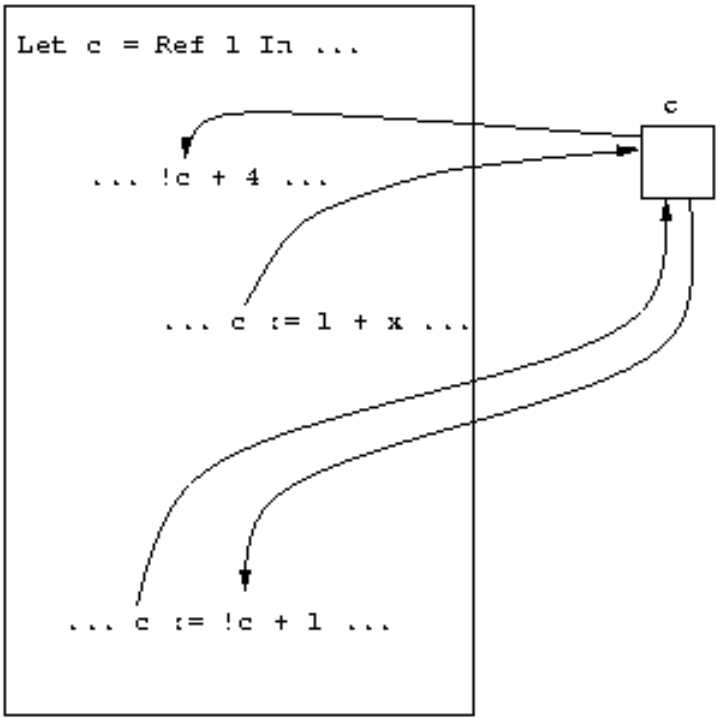


Figure 4.1: The "junction box" view of reference cells.

cells is to think of each one as a junction box. Then, each assignment is a path into the junction, and each read, or dereference, is a path going out of the junction. The reference cell, or junction, sits to the side of the program, and allows distant parts of the program to communicate with one another. This "distant communication" can be a useful programming paradigm, but again it should be used sparingly. Figure 4.1 illustrates this metaphor.

In the world of C++ and Java, non-const (or non-final) global variables are the most notorious form of reference. While globals make it easy to do certain tasks, they generally make programs difficult to debug, since they can be altered anywhere in the code.

4.1.1 The F♭S Language

Adding state to our language involves making significant modifications to the pure functional languages that we've looked at so far. Therefore, we will spend a bit of time developing a solid operational semantics for our state-based language before looking at the interpreter. We will define a language **F♭S**: **F♭** with state.

The most significant departure from the pure functional operational semantics we have looked at so far is that the evaluation relation $e \Rightarrow v$ is not sufficient to capture the semantics of a state-based language. The evaluation of an expression can now produce side-effects, and our $\Rightarrow$ rule needs to incorporate this somehow. Specifically, we will need

to have someplace to record all these side-effects: a **store**. In C, a store is a stack and a heap, and memory locations are referenced by their addresses. When we are writing our interpreter, we will only have access to the heap, so we will need to create an abstract store in which to record side-effects.

Definition 4.1 (Store)**.** *A **store** is a finite map $c \mapsto v$ of cell names to values.*

Cell names are an abstract representation of a memory location. A store is therefore an abstraction of a runtime heap. The C heap is a low-level realization of a store where the cell names are simply numerical memory addresses. A store is also known as a *dictionary* from a data-structures perspective. We write $\mathrm{Dom}(S)$ to refer the domain of this finite map, that is, the set of all cells that it maps.

Let us begin by extending the concrete syntax of **F♭** to include **F♭S** expressions. The additions are as follows.

- Referencing, `Ref` e.
- Assignment, e `:=` e'.
- Dereferencing, `!`e.
- Cell names, c.

We need cell names because `Ref 5` needs to evaluate to a location, c, in the store. Because cell names refer to locations in the heap, and the heap is initially empty, programs have no cell names when the execution begins.

Although not part of the **F♭S** syntax, we will need a notation to represent operations on the store. We write $S\{c \mapsto v\}$ to indicate the store S extended or modified to contain the mapping $c \mapsto v$. We write $S(c)$ to denote the value of cell c in store S.

Now that we have developed the notion of a store, we can define a satisfactory evaluation relation for **F♭S**. Evaluation is written as follows.

$$\langle e, S_0 \rangle \Rightarrow \langle v, S \rangle,$$

where at the start of computation, S_0 is initially empty, and where S is the final store when computation terminates. In the process of evaluation, cells, c, will begin to appear in the program syntax, as references to memory locations. Cells are values since they do not need to be evaluated, so the value space of **F♭S** also includes cells, c.

Evaluation Rules for F♭S

Finally, we are ready to write the evaluation rules for **F♭S**. We will need to modify all of the **F♭** rules with the store in mind (recall that our evaluation rule is now $\langle e, S_0 \rangle \Rightarrow \langle v, S \rangle$, not simply $e \Rightarrow v$). We do this by **threading** the store along the flow of control. Doing this introduces a great deal more dependency between the rules, even the ones that do not directly manipulate the store. We will rewrite the function application rule for **F♭S** to illustrate the types of changes that are needed in the other rules.

(Function Application)

$$\frac{\langle e_1, S_1\rangle \Rightarrow \langle \texttt{Function } x \texttt{ -> } e, S_2\rangle, \quad \langle e_2, S_2\rangle \Rightarrow \langle v_2, S_3\rangle, \quad \langle e[v_2/x], S_3\rangle \Rightarrow \langle v, S_4\rangle}{\langle e_1\ e_2, S_1\rangle \Rightarrow \langle v, S_4\rangle}$$

Note how the store here is *threaded* through the different evaluations, showing how changes in the store in one place propagate to the store in other places, and in a fixed order that reflects the indented evaluation order. The rules for our new memory operations are as follows.

(Reference Creation)
$$\frac{\langle e, S_1\rangle \Rightarrow \langle v, S_2\rangle}{\langle \texttt{Ref } e, S_1\rangle \Rightarrow \langle c, S_2\{c \mapsto v\}\rangle, \text{ for } c \notin \mathrm{Dom}(S_2)}$$

(Dereference)
$$\frac{\langle e, S_1\rangle \Rightarrow \langle c, S_2\rangle}{\langle \texttt{!}e, S_1\rangle \Rightarrow \langle v, S_2\rangle, \text{ where } S_2(c) = v}$$

(Assignment)
$$\frac{\langle e_1, S_1\rangle \Rightarrow \langle c, S_2\rangle, \quad \langle e_2, S_2\rangle \Rightarrow \langle v, S_3\rangle}{\langle e_1 \texttt{ := } e_2, S_1\rangle \Rightarrow \langle v, S_3\{c \mapsto v\}\rangle}$$

These rules can be tricky to evaluate because the store needs to be kept up to date at all points in the evaluation. Let us look at a few example expressions to get a feel for how this works.

Example 4.1.

```
!(!(Ref Ref 5)) + 4
```

$\langle$`!(!(Ref Ref 5)) + 4`$\rangle, \{\} \Rightarrow \langle$`9`$, \{c_1 \mapsto$ `5`$, c_2 \mapsto c_1\}\rangle$, because
  $\langle$`!(!(Ref Ref 5))`$, \{\}\rangle \Rightarrow \langle$`5`$, \{c_1 \mapsto$ `5`$, c_2 \mapsto c_1\}\rangle$, because
    $\langle$`!(Ref Ref 5)`$, \{\}\rangle \Rightarrow \langle c_1, \{c_1 \mapsto$ `5`$, c_2 \mapsto c_1\}\rangle$, because
      $\langle$`Ref Ref 5`$, \{\}\rangle \Rightarrow \langle c_2, \{c_1 \mapsto$ `5`$, c_2 \mapsto c_1\}\rangle$, because
        $\langle$`Ref 5`$, \{\}\rangle \Rightarrow \langle c_1, \{c_1 \mapsto$ `5`$\}\rangle$

Example 4.2.

```
(Function y -> If !y = 0 Then y Else 0) Ref 7
```

$\langle$`(Function y -> ...) Ref 7`$, \{\}\rangle \Rightarrow \langle$`0`$, \{c_1 \mapsto$ `7`$\}\rangle$, because
  $\langle$`Ref 7`$, \{\}\rangle \Rightarrow \langle c_1, \{c_1 \mapsto$ `7`$\}\rangle$, and
  $\langle$`(If !y = 0 Then y Else 0)`$[c_1/$`y`$], \{c_1 \mapsto$ `7`$\}\rangle \Rightarrow$
    $\langle$`0`$, \{c_1 \mapsto$ `7`$\}\rangle$, because
    $\langle$`!`c_1 `= 0`$, \{c_1 \mapsto$ `7`$\}\rangle \Rightarrow \langle$`False`$, \{c_1 \mapsto$ `7`$\}\rangle$, because
      $\langle$`!`$c_1, \{c_1 \mapsto$ `7`$\}\rangle \Rightarrow \langle$`7`$, \{c_1 \mapsto$ `7`$\}\rangle$

F♭S Interpreters

Just as we had to modify the evaluation relation in our **F♭S** operational semantics to support state, writing a **F♭S** interpreter will also require some additional work. There are two obvious approaches to take, and we will treat them both below. The first approach involves mimicking the operational semantics and defining evaluation on an expression and a store together. This approach yields a *functional* interpreter in which `eval`(e, S_0) for expression e and initial state S_0 returns the tuple (v, S), where v is the resulting value and S is the final state.

The second and more efficient design involves keeping track of state in a global, mutable dictionary structure. This is generally how real implementations work. This approach results in more familiar evaluation semantics, namely `eval` e returns v. Obviously such an interpreter is no longer functional, but rather, *imperative*. We would like such an interpreter to faithfully implement the operational semantics of **F♭S**, and thus we would ideally want a theorem that states that this approach is equivalent to the first approach. Proving such a theorem would be difficult, however, mainly because our proof would rely on the operational semantics of OCaml, or whatever implementation language we chose. We will therefore take it on good faith that the two approaches are indeed equivalent.

The Functional Interpreter The functional interpreter implements a *stateful* language in a *functional* way. It threads the state through the evaluation, just as we did when we defined our operational semantics. Imperative style programming can be "hacked" into a functional style by threading the state along, and there are regular methods for threading state through any functional programs, namely *monads*. The threading approach to modeling state functionally was first employed by Strachey in the late 1960's [22]. Note that an operational semantics is *always* pure functional, since mathematics is always purely functional.

To implement the functional interpreter, we model the store using a finite mapping of integer keys to values. The skeleton of the implementation is presented below.

```
(* Declare all the expr, etc types globally for convenience. *)

(* The store functionality is a separate module.  *)

module type STORE =
  sig
   (* ... *)
  end

(* The Store structure implements a (functional) store.  A simple
 * implementation could be via a list of pairs such as
 *
 *   [((Cell 2),(Int 4)); ((Cell 3),Plus((Int 5),(Int 4))); ... ]
 *)

module Store : STORE =
```

```
type store = (* ... *)

 struct
  let empty = (* initial empty store *)
  let fresh = (* returns a fresh Cell name *)
    let count = ref 0 in
    function () -> ( count := !count + 1; Cell(!count) )
  (* Note: this is not purely functional!  It is quite
   * difficult to make fresh purely functional.
   *)

(* Look up value of cell c in store s *)
   let lookup (s,c) = (* ... *)

(* Add or modify cell c to point to value v in the store s.
 * Return the new store.
 *)
   let modify(s,c,v) =  (* ... *)
  end

(* The evaluator is then a functor taking a store module *)

module FbSEvalFunctor =
  functor (Store : STORE) ->
  struct

    (* ... *)


    let eval (e,s) = match e with
      (Int n) -> ((Int n),s) (* values don't modify store *)
    | Plus(e,e') ->
        let (Int n,s') = eval(e,s) in
        let (Int n',s'') = eval(e',s') in
        (Int (n+n'),s'')

(* Other cases such as application use a similar store
 * threading technique.
 *)
    | Ref(e) -> let (v,s') = eval(e,s) in
                let c = Store.fresh() in
                   (c,Store.modify(s',c,v))
    | Get(e) -> let (Cell(n),s') = eval(e,s) in
      (Store.lookup(Cell(n)),s')
    | Set(e,e') ->  (* exercise *)
```

```
  end

module FbSEval = FbSEvalFunctor(Store)
```

The Imperative Interpreter The stateful, imperative **F♭S** interpreter is more efficient than its functional counterpart, because no threading is needed. In the imperative interpreter, we represent the store as a dictionary structure (similar to Java's `HashMap` class or the C++ STL `map` template). The `eval` function needs no extra store parameter, provided that it has a reference to the global dictionary. Non-store-related rules such as `Plus` and `Minus` are completely ignorant of the store. Only the direct store evaluation rules, `Ref`, `Set`, and `Get` actually extend, update, or query the store. A good evaluator would also periodically garbage collect, or remove unneeded store elements. Garbage collection is discussed briefly in Section 4.1.4. The implementation of the stateful interpreter is left as an exercise to the reader.

Side-Effecting Operators

Now that we have a mutable store, our code has properties beyond the value that is returned, namely, side-effects. Operators such as sequencing (`;`), and `While`- and `For` loops now become relevant. These syntactic concepts are easily defined as macros, so we will not add them to the official **F♭S** syntax. The macro definitions are as follows.

e_1`;` e_2 = `(Function x ->` e_2`)` e_1
`While` e `Do` e' = `Let Rec f x = If` e `Then f` e' `Else 0 In f 0`

Exercise 4.1. Why are sequencing and loop operations irrelevant in pure functional languages like **F♭**?

4.1.2 Cyclical Stores

An interesting phenomenon is possible with stateful languages. It is possible to make a **cyclical store**, that is, a cell whose contents are a pointer to itself. Creating a cyclic store is quite easy:

```
Let x = Ref 0 in x := x
```

This is the simplest possible store cycle, where a cell directly points to itself. This type of store is illustrated in Figure 4.2.

Exercise 4.2. In the above example, what does `!!!!!!!!x` return? Can a store cycle like the one above be written in OCaml? Why or why not? (**Hint**: What is the type of such an expression?)

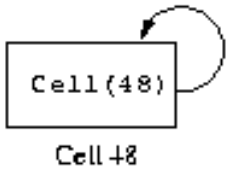


Figure 4.2: A simple cycle.

A more subtle form of a store cycle is when a function is placed in the cell, and the body of the function refers to that cell. Consider the following example.

```
Let c = Ref 0 In
   c := (Function x -> If x = 0 Then 0 Else 1 + !c(x-1));
!c(10)
```

Cell `c` contains a function, which, in turn, refers to `c`. In other words, the function has a reference to itself, and can apply itself, giving us recursion. This form of recursion is known as **tying the knot**, and is the method used by most compilers to implement recursion. A similar technique is used to make objects self-aware, although C++ explicitly passes self, and is more like the Y-combinator.

Exercise 4.3. Tying the knot can be written in OCaml, but not directly as above. How must the reference be declared for this to work? Why can we create this sort of cyclic store, but not the sort described in Exercise 4.2?

Interact with F♭SR. Let's write a recursive multiplication function in **F♭SR**, first using `Let Rec`, and then by tying the knot.

```
# Let Rec mult x = Function y ->
    If x = 0 Then
      0
    Else
      y + mult (x - 1) y In
  mult 8 9;;
==> 72
```

Now we'll use tying the knot. Because **F♭SR** does not include a sequencing operation, we use the encoding presented in Section 4.1.1.

```
# Let mult = Ref 0 In
  (Function dummy -> (!mult) 9 8)
    (mult := (Function x -> Function y ->
       If x = 0 Then
         0
       Else
         y + (!mult) (x - 1) y));;
==> 72
```

4.1.3 The "Normal" Kind of State

Languages like C++, Java, and Scheme have a different form for expressing mutation. There is no need for an explicit dereference (`!`) operator to get the value of a cell. This form of mutation is more difficult to capture in an interpreter, because variables mean something different depending on whether they are on the left or right side of an assignment operator. An **l-value** occurs on the left side of the assignment, and represents a memory location. An **r-value** to the right of an assignment, or elsewhere in the code, and represents an actual value.

Consider the C/C++/Java assignment statement `x = x + 1`. The `x` on the left of the assignment is an l-value, and the `x` in `x + 1` is an r-value. In OCaml, we would write a similar expression as `x := !x + 1`. Thus, OCaml is explicit in which is being referred to, the cell or the value. For a Java l-value, the cell is needed to perform the store operation, and for a Java r-value, the *value* of the cell is needed, which is why, in OCaml, we must write `!x`.

l-values and r-values are distinct. Some expressions may be both l-values and r-values, such as `x` and `x[3]`. Other values may only be r-values, such as `5`, `(0 == 1)`, and `sin(3.14159)`. Therefore, l-values and r-values are expressed differently in the language grammar, specifically, l-values are a subset of r-values. There are some shortcomings to such languages. For example, in OCaml we can say `f(3) := 1`, which assigns the value `1` to the cell returned by the function `f`. Expressions of this sort are invalid in Java, because `f(3)` is not an l-value. We could revise the **F♭S** grammar to use this more restrictive notion, where l-values must only be variables, in the following way:

```
type expr =
  ...
| Get of expr | Set of ident * expr | Ref of expr
```

For the variable on the left side of the assignment, we would need the address, and not the contents, of the variable.

A final issue with the standard notion of state in C and C++ is the problem of uninitialized variables. Because variables are not required to be initialized, runtime errors due to these uninitialized variables are possible. Note that the `Ref` syntax of **F♭S** and OCaml requires that the variable be explicitly initialized as part of its declaration.

4.1.4 Automatic Garbage Collection

Memory that is allocated may also, at some point, be freed. In C and C++, this is done explicitly with `free()` and `delete()` respectively. However, Java, OCaml, Smalltalk, Scheme, and Lisp, to name a few, support the automated freeing of unused memory through a **garbage collector**.

Garbage collection is a large area of research (see [23] for a survey), but we will briefly sketch out what an implementation looks like.

First, something triggers the garbage collector. Generally, the trigger is that the store is getting too full. In Java, the garbage collector may be explicitly invoked by the method `System.gc()`. When the garbage collector is invoked, evaluation is suspended until the garbage collector has completed. The garbage collector proceeds to scan through the

current computation to find cells that are directly used. The set of these cells is known as the *root set*. Good places to look for roots are on the evaluation stack and in global variable locations.

Once the root set is established, the garbage collector marks all cells not in the root set as initially *free*. Then, it recursively traverses the memory graph starting at the root set, and marks cells that are reachable from the root set as *not free*, thus at the end of the traversal, all cells that are in a different connected component from *any* of the cells in the root set are marked *free*, and these cells may be reused. There are many different ways to reuse memory, but we will not go into detail here.

4.2 Environment-Based Interpreters

Now that we have discussed stateful languages, we are going to briefly touch on some efficiency issues with the way we have been defining interpreters. We will focus on eliminating explicit substitutions in function application.

A "low-level" interpreter would never duplicate the function argument for each variable usage. The problem with doing so is that it can severely increase the size of the data that must be maintained by the interpreter. Consider an expression in the following form.

```
(Function x -> x x x) (whopping expr)
```

This expression is evaluated by computing

```
(whopping expr)(whopping expr)(whopping expr),
```

tripling the size of the data.

In order to solve this problem, we define an **explicit environment interpreter**. In an explicit environment interpreter, instead of doing direct variable substitution, we keep track of each variable's value in a **runtime environment**. The runtime environment is simply a mapping of variables to values. We write environments as $\{x_1 \mapsto v_1, x_2 \mapsto v_2\}$, to mean that variable x_1 maps to value v_1 and variable x_2 maps to value v_2.

Now, to compute

```
(Function x -> x x x)(whopping expr)
```

the interpreter computes `(x x x)` in the environment $\{\texttt{x} \mapsto \text{whopping expr}\}$.

Technically, even the simple substituting interpreters we've looked at don't really make three copies of the data. Rather, they maintain a single copy of the data and three pointers to it. This is because immutable data can always be passed by reference since

it never changes. However, in a compiler the code can't be copied around like this, so a scheme like the one presented above is necessary.

There is a possibility for some anomalies with this approach, however. Specifically, there is a problem when a function is returned as the result of another function application, and the returned function has local variables in it. Consider the following example.

```
f = Function x -> If x = 0 Then
   Function y -> y
Else Function y -> x + y
```

When f `(3)` is computed, the environment binds `x` to `3` while the body is computed, so the result returned is

```
Function y -> x + y,
```

but it would be a mistake to simply return this as the correct value, because we would lose the fact that `x` was bound to `3`. The solution is that when a function value is returned, the *closure* of that function is returned.

Definition 4.2 (Closure)**.** *A **closure** is a function along with an environment, such that all free values in the function body are bound to values in the environment.*

For the case above, the correct value to return is the closure

```
(Function y -> x + y, {x ↦ 3})
```

Theorem 4.1. *A substitution-based interpreter and an explicit environment interpreter for **F♭** are equivalent: all **F♭** programs either terminate on both interpreters or compute forever on both interpreters.*

This closure view of function values is critical when writing a compiler, since compiler's should not be doing substitutions of code on the fly. Compilers are discussed in Chapter 8.

4.3 The F♭SR Language

We can now define the **F♭SR** language. **F♭SR** is the call-by-value language that included the basic features of **F♭**, and is extended to include records (**F♭R**), and state (**F♭S**).

In Section 4.4 and Chapters 5 and 6, we will study the language features missing from **F♭SR**, namely objects, exceptions, and types. In Chapter 8, we will discuss translations for **F♭SR**, but will not include these other language features. The abstract syntax type for **F♭SR** is as follows.

```
type label = Lab of string

type ident = Ident of string

type expr =
    Var of ident | Function of ident * expr | Appl of expr * expr |
    LetRec of ident * ident * expr * expr | Plus of expr * expr |
    Minus of expr * expr | Equal of expr * expr |
    And of expr * expr | Or of expr * expr | Not of expr |
    If of expr * expr * expr | Int of int | Bool of bool |
    Ref of expr | Set of expr * expr | Get of expr | Cell of int |
    Record of (label * expr) list | Select of  label * expr |
    Let of ident * expr * expr

type fbtype =
    TInt | TBool | TArrow of fbtype * fbtype | TVar of string |
    TRecord of (label * fbtype) list | TCell of fbtype;;
```

In the next two sections we will look at some nontrivial "real-world" **F♭SR** programs to illustrate the power we can get from this simple language. We begin by considering a function to calculate the factorial function, and conclude the chapter by examining an implementation of the merge sort algorithm.

4.3.1 Multiplication and Factorial

The factorial function is fairly easy to represent using **F♭SR**. The main point of interest here is the encoding of multiplication using a curried `Let Rec` definition. This example assumes a positive integer input.

```
(*
 * First we encode multiplication for positive nonnegative
 * integers.  Notice the currying in the Let Rec construct.
 * Multiplication is encoded in the obvious way: repeated
 * addition.
 *)
Let Rec mult x = Function y ->
  If y = 0 Then
    0
  Else
    x + (mult x (y - 1)) In

(*
 * Now that we have multiplication, factorial is easy to
 * define.
 *)
Let Rec fact x =
  If x = 0 Then
```

```
    1
  Else
    mult x (fact (x - 1)) In

fact 7
```

4.3.2 Merge Sort

Writing a merge sort in **F♭SR** is a fairly comprehensive example. One of the biggest challenges is encoding integer comparisons, i.e. $<$, $>$, etc. Let's discuss how this is accomplished before looking at the code.

First of all, given that we already have an equality test, `=`, encoding the `<=` operation basically gives us the other standard comparison operations for free. Assuming that we have a `lesseq` function, the other operations can be trivially encoded as follows.

```
Let lesseq = (* real definition *) In

Let lessthan = (Function x -> Function y ->
  (lesseq x y) And Not (x = y)) In

Let greaterthan = (Function x -> Function y ->
  Not (lesseq x y)) In

Let greatereq = (Function x -> Function y ->
  (greaterthan x y) Or (x = y)) In ...
```

Therefore it suffices to encode `lesseq`. But how can we do this using only the regular **F♭SR** operators? The basic idea is as follows. To test if x is less than or equal to y we compute a z such that $x + z = y$. If z is nonnegative, we know that x is less than or equal to y. If z is negative, we know that x is greater than y.

At first glance, we seem to have a "chicken and egg" problem here. How can we tell if z is negative if we have no comparison operator. And how do we actually compute z to begin with? One idea would be to start with $z = 0$ and loop, incrementing z by 1 at every step, and it testing if $x + z = y$. If we find a z, we stop. We know that z is positive, and we conclude that x is less than or equal to y. If we don't find a z, we start at $z = -1$ and perform a similar loop, decrementing z by 1 every step. If we find a proper value of z this way, we conclude that x is greater then y.

The flaw in the above scheme should be obvious: if $x > y$, the first loop will never terminate. Indeed we would need to run both loops in parallel for this idea to work! Obviously **F♭SR** doesn't allow for parallel computation, but there is a solution along these lines. We can interleave the values of z that are tested by the two loops. That is, we can try the values $\{0, -1, 1, -2, 2, -3, \ldots\}$.

The great thing about this approach is that every other value of z is negative, and we can simply pass along a boolean value to represent the sign of z. If z is nonnegative, the next iteration of the loop inverts it and subtracts 1. If z is negative, the next iteration

simply inverts it. Note that we can invert the sign of an number `x` in **F♭SR** by simply writing `0 - x`. Now, armed with our ideas about `lesseq`, let us start writing our merge sort.

```
(* We need to represent the empty list somehow.  Since we
 * will need to test for it, and since equality is only
 * defined on integers, emptylist will need to be an
 * integer.  We define it as -1.
 *)
Let emptylist = (0 - 1) In


(* Next, let's define some list operations, head, tail,
 * cons, and length.  There encodings are straightforward,
 * and were covered in the text.  Notice that we are
 * encoding lists as records, using {l,r} records like
 * pairs.
 *)
Let head = Function seq -> seq.l In

Let tail = Function seq -> seq.r In

Let cons = Function elt -> Function seq -> {l=elt; r=seq} In

Let Rec length seq =
   If seq = emptylist Then
     0
   Else
     1 + length (seq.r) In


(* Now, we're ready to define lesseq.  Notice how lesseq is
 * a wrapper function that uses le.  le passes along the
 * sign of v as an argument, as discussed in the text.
 *)
Let lesseq = Function a -> Function b ->
  Let Rec le x =
    Function y -> Function v ->
      Function v_is_non_neg ->
        If (x + v) = y Then
          v_is_non_neg
        Else
          If v_is_non_neg Then
            le x y (0 - v - 1) (Not v_is_non_neg)
          Else
            le x y (0 - v) (Not v_is_non_neg) In
```

```
  le a b 0 True In

(* This function takes a list and splits it into nearly
 * equal halves.  These halves are returned as a pair
 * (encoded as an {left, right} record).
 *)
Let split = Function seq ->
  Let Rec splt seq1 = Function seq2 ->
    If lesseq (length seq1) (length seq2) Then
      {left=seq1; right=seq2}
    Else
      splt (tail seq1) (cons (head seq1) seq2) In
  splt seq emptylist In

(* Here is where we merge two sorted lists.  We scan through
 * each list in parallel, chopping off the smaller of the
 * two list heads and appending it to the result.  This is
 * where we make use of our lesseq function.
 *)
Let Rec merge seq1 = Function seq2 ->
  If seq1 = emptylist Then
    seq2
  Else If seq2 = emptylist Then
    seq1
  Else
    If lesseq (head seq1) (head seq2) Then
      cons (head seq1) (merge (tail seq1) seq2)
    Else
      cons (head seq2) (merge seq1 (tail seq2)) In

(* mergesort sorts a single list by breaking it into two
 * smaller lists, recursively sorting those lists, and
 * merging the two back into a single list.  Recall that
 * the base cases of the recursion are a single element list
 * and an empty list, both of which are necessarily in
 * sorted order by definition.
 *)
Let Rec mergesort seq =
  If lesseq (length seq) 1 Then
    seq
  Else
    Let halves = split seq In
    merge (mergesort (halves.left))
          (mergesort (halves.right)) In

(* Finally we call mergesort on an actual list.  Notice the
```

```
 * record encoding.
 *)
mergesort {l=5;  r=
          {l=6;  r=
          {l=2;  r=
          {l=1;  r=
          {l=4;  r=
          {l=7;  r=
          {l=8;  r=
          {l=10; r=
          {l=9;  r=
          {l=3;  r=emptylist}}}}}}}}}}
```

4.4 Exceptions and Other Control Operations

Until now, expressions have been evaluated by marching through the evaluation rules one at a time. To perform addition, the left and right operands were evaluated to values, after which the addition expression itself was evaluated to a value. This addition expression may have been part of a larger expression, which could then itself have been evaluated to a value, etc.

In this section, we will discuss explicit **control operations**, concentrating on exceptions. Explicit control operations are operations that explicitly alter the control of the evaluation process. Even simple languages have control operations. A common example is the `return` statement in C++ and Java.

In **F**♭, the value of the function is whatever its entire body evaluates to. If, in the middle of some complex conditional loop expression, we have the final result of the computation, it is still necessary to complete the execution of the function. A return statement gets around this problem by immediately returning from the function and *aborting* the rest of the function computation.

Another common control operation is the loop exit operation, or `break` in C++ or Java. `break` is similar to `return`, except that it exits the current loop instead of the entire function.

These types of control operations are interesting, but in this section, we will concentrate more on two more powerful control operations, namely **exceptions** and **exception handlers**. The reader should already be familiar with the basics of exceptions from working with the OCaml exception mechanism.

There are some other control operations that we will not discuss, such as the call/cc, shift/reset, and control/prompt operators.

We will avoid the `goto` operator in this section, except for the following brief discussion. `goto` basically jumps to a labeled position in a function. The main problem with `goto` is that it is too raw. The paradigm of jumping around between labels is not all that useful in the context of functions. It is also inherently dangerous, as one may inadvertently jump into the middle of a function that is not even executing, or skip past variable

initializations. In addition, with a rich enough set of other control operators, `goto` really doesn't provide any more expressiveness, at least not meaningful expressiveness.

The truth is that control operators are really not needed at all. Recall that **F**♭, and the lambda calculus for that matter, are already Turing-complete. Control operators are therefore just conveniences that make programming easier. It is useful to think of control operators as "meta-operators," that is, operators that act on the evaluation process itself.

4.4.1 Interpreting Return

How are exceptions interpreted? Before we answer this question, we will consider adding a `Return` operator to **F**♭, since it is a simpler control operator than exceptions. The trouble with `Return`, and with other control operators, is that it doesn't fit into the normal evaluation scheme. For example, consider the expression

```
(Function x ->
   (If x = 0 Then 5 Else Return (4 + x)) - 8) 4
```

Since `x` will not be `0` when the function is applied, the `Return` statement will get evaluated, and execution should stop immediately, not evaluating the "`- 8`." The problem is that evaluating the above statement means evaluating

```
(If 4 = 0 Then 5 Else Return (4 + 4)) - 8,
```

which, in turn, means evaluating

```
(Return (4 + 4)) - 8.
```

But we know that the subtraction rule works by evaluating the left and right hand sides of this expression to values, and performing integer subtraction on them. Clearly that doesn't work in this case, and so we need a special rules for subtraction with a `Return` in one of the subexpressions.

First, we need to add `Return`s to the value space of **F**♭ and provide an appropriate evaluation rule:

$$\textit{(Return)} \qquad \frac{e \Rightarrow v}{\texttt{Return}\ e \Rightarrow \texttt{Return}\ v}$$

Next, we need the special subtraction rules, one for when the `Return` is on the left side, and one for when the `Return` is on the right side.

(- Return Left) $$\frac{e \Rightarrow \texttt{Return}\ v}{e\ \texttt{-}\ e' \Rightarrow \texttt{Return}\ v}$$

(- Return Right) $$\frac{e \Rightarrow v, \quad e' \Rightarrow \texttt{Return}\ v'}{e\ \texttt{-}\ e' \Rightarrow \texttt{Return}\ v'} v \text{ is not of the form } \texttt{Return}\ v''$$

Notice that these subtraction rules evaluate to `Return` v and not simply v. This means that the `Return` is "bubbled up" through the subtraction operator. We need to define similar return rules for *every* **F♭** operator. Using these new rules, it is clear that

```
Return (4 + 4) - 8 ⇒ Return 8.
```

Of course, we don't want the `Return` to bubble up indefinitely. When the `Return` pops out of the function application, we only get the value associated with it. In other words, our original expression,

```
(Function x ->
   (If x = 0 Then 5 Else Return (4 + x)) - 8) 4
```

should evaluate to `8`, not `Return 8`. To accomplish this, we need a special function application rule.

(Appl. Return) $$\frac{e_1 \Rightarrow \texttt{Function}\ x\ \texttt{->}\ e, \quad e_2 \Rightarrow v_2, \quad e[v_2/x] \Rightarrow \texttt{Return}\ v}{e_1\ e_2 \Rightarrow v}$$

A few other special application rules are needed for the cases when either the function or the argument itself is a `Return`.

(Appl. Return Function) $$\frac{e_1 \Rightarrow \texttt{Return}\ v}{e_1\ e_2 \Rightarrow \texttt{Return}\ v}$$

(Appl. Return Arg.) $$\frac{e_1 \Rightarrow v_1,\ e_2 \Rightarrow \texttt{Return}\ v}{e_1\ e_2 \Rightarrow \texttt{Return}\ v}$$

Of course, we still need the original function application rule (see Section 2.3.3) for the case that function execution implicitly returns a value by dropping off the end of its execution.

Let us conclude our discussion of `Return` by considering the effect of `Return Return` e. There are two possible interpretations for such an expression. By the above rules, this expression returns from two levels of function calls. Another interpretation would be to add the following rule:

(Double Return) $$\frac{e \Rightarrow \texttt{Return}\ v}{\texttt{Return}\ e \Rightarrow \texttt{Return}\ v}$$

Of course we would need to restrict the original `Return` rule to the case where v was not in the form `Return` v. With this rule, instead of returning from two levels of function calls, the second `Return` actually interrupts and bubbles through the first. Of course, double returns are not a common construct, and these rules will not be used often in practice.

4.4.2 The F♭X Language

The translation of `Return` that was given above can easily be extended to deal with general exceptions. We will define a language **F♭X**, which is **F♭** extended with a OCaml-style exception mechanism. **F♭X** does not have `Return` (nor does OCaml), but `Return` is easily encodable with exceptions. For example, the "pseudo-OCaml" expression

```
(function x -> (if x = 0 then 5 else return (4 + x)) - 8) 4
```

can be encoded in the following manner.

```
exception Return of int;;

(function x ->
  try
    (if x = 0 then 5 else raise (Return (4 + x))) - 8
  with
    Return n -> n) 4;;
```

`Return` can be encoded in other functions in a similar manner.

Now, let's define our **F♭X** language. The basic idea is very similar to the `Return` semantics that we discussed above. We define a new kind of value,

`Raise #`*xn* v,

which bubbles up the exception `#`*xn*. This is the generalization of the value class `Return` v from above. `#`*xn* is a metavariable representing an exception name. An exception contains two pieces of data: a name, and an argument. The argument can be an arbitrary expression. Although we only allow single-argument exceptions, zero-valued or multi-valued versions of exceptions can be easily encoded by, for example, supplying the exception with a record argument with zero, one, or several fields. We also add to **F♭X** the expression

`Try` e `With #`*xn* x `->` e'

Note that `Try` binds free occurrences of x in e'. Also notice that the **F♭X** `Try` syntax differs slightly from the OCaml syntax in that OCaml allows an arbitrary pattern-match expression in the `With` clause. We allow only a single clause that matches all values of `#`x.

F♭X is untyped, so exceptions do not need to be declared. We use the "`#`" symbol to designate exceptions in the concrete syntax, for example, `#MyExn`. Below is an example of a **F♭X** expression.

```
Function x -> Try
   (If x = 0 Then 5 Else Raise #Return (4 + x)) - 8
With #Return n -> n) 4
```

Exceptions are side-effects, and can cause "action at a distance." Therefore, like any other side-effects, they should be used sparingly.

4.4.3 Implementing the F♭X Interpreter

The abstract syntax type for **F♭X** is as follows.

```
type exnlab = string
type expr =
  ...
| Raise of exnlab * expr
| Try of expr * exnlab * ident * expr
```

The rules for `Raise` and `Try` are derived from the return rule and the application return rule respectively. Raise "bubbles up" an exception, just like `Return` bubbled itself up. `Try` is the point at which the bubbling stops, just like function application was the point at which a `Return` stopped bubbling. The operational semantics of exceptions are as follows.

$$\textit{(Raise)} \quad \frac{e \Rightarrow v, \text{ for } v \text{ not of the form } \texttt{Raise } \ldots}{\texttt{Raise \#}xn\ e \Rightarrow \texttt{Raise \#}xn\ v}$$

$$\textit{(Try)} \quad \frac{e \Rightarrow v \text{ for } v \text{ not of the form } \texttt{Raise \#}xn\ v}{\texttt{Try } e \texttt{ With \#}xn\ x \texttt{ -> } e' \Rightarrow v}$$

$$\textit{(Try Catch)} \quad \frac{e \Rightarrow \texttt{Raise \#}xn\ v, \quad e'[v/x] \Rightarrow v'}{\texttt{Try } e \texttt{ With \#}xn\ x \texttt{ -> } e' \Rightarrow v'}$$

In addition, we need to add special versions of all of the other **F♭** rules so that the `Raise` bubbles up through the computation just as the `Return` did. For example

$$\textit{(- Raise Left)} \quad \frac{e \Rightarrow \texttt{Raise \#}xn\ v}{e \texttt{ - } e' \Rightarrow \texttt{Raise \#}xn\ v}$$

Note that we must handle the unusual case of when a `Raise` bubbles through a `Raise`, something that will not happen very often in practice. The rule is very much like the "`-` Raise Left" rule above.

(Raise Raise) $$\frac{e \Rightarrow \texttt{Raise \#}xn\ v}{\texttt{Raise}\ e \Rightarrow \texttt{Raise \#}xn\ v}$$

Now, let's trace through the execution of

```
Function x -> Try
      (If x = 0 Then 5 Else Raise #Return (4 + x)) - 8
   With #Return n -> n) 4
```

After the function application and the evaluation of the `If` statement, we are left with

```
Try (Raise #Return (4 + 4)) - 8
   With #Return n -> n
```

which is

```
Try (Raise #Return 8) - 8
   With #Return n -> n
```

which by bubbling in the subtraction computes to

```
Try Raise #Return 8
   With #Return n -> n
```

which by the Try Catch rule returns the value `8`, as expected.

Chapter 5

Object-Oriented Language Features

Object-oriented programming has become an industry standard. Yet, object-oriented features do not fundamentally add much to a language. They certainly do not lead to shorter programs. The success of object-oriented programming is due mainly to the fact that it is a style that is appropriate for the human psychology. Humans, as part of their basic function, are highly adept at recognizing and interacting with everyday objects. And, objects in programming are enough like objects in the everyday world that our rich intuitions can apply to them.

Before we discuss the properties of objects in object-oriented programming, let us briefly review some of the more important properties of *everyday* objects that would make them useful for programming.

- Everyday objects are *active*, that is, they are not fully controlled by us. Objects have internal and evolving *state*. For instance, a running car is an active object, and has a complex internal state including the amount of gas remaining, the engine coolant and transmission fluid levels, the amount of battery power, the oil level, the temperature, the degree of wear and tear on the engine components, etc.

- Everyday objects are *communicative*. We can send messages to them, and we can get feedback from objects as a result of sending them messages. For example, starting a car and checking the gas gauge can both be viewed as sending messages to the car.[1]

- Everyday objects are *encapsulated*. They have internal properties that we can not see, although we can learn some of them by sending messages. To continue the car example, checking the gas gauge tells us how much gas is remaining, even though we can't see into the gas tank directly, and we don't know if it contains regular or premium gas.

[1] It may, at first, seem unnatural to view checking the gas gauge as sending a message. After all, it is really the car sending a message to us. We view ourselves as the "sender," however, because we *initiated* the check, that is, in a sense, we *asked* the car how much gas was remaining.

- Everyday objects may be *nested*, that is, objects may be made up of several smaller object components, and those components may themselves have smaller components. Once again, a car is a perfect example of a nested object, being made of of smaller objects including the engine and the transmission. The transmission is also a nested object, including an input shaft, an output shaft, a torque converter, and a set of gears.
- Everyday objects are, for the most part, uniquely named. Cars are uniquely named by license plates or vehicle registration numbers.
- Everyday objects may be *self-aware*, in the sense that they may intentionally interact with themselves, for instance a dog licking his paw.
- Everyday object interactions may be *polymorphic* in that a diverse set of objects may share a common messaging protocol, for example the accelerator/brake/steering wheel messaging system common to all models of cars and trucks.

The objects in object-oriented programming also have these properties. For this reason, object-oriented programming has a natural and familiar feel to most people. Let us now consider objects of the programming variety.

- Objects have an internal state in the form of instance variables or fields. Objects are generally *active*, and their state is not fully controlled by their callers.
- Objects support messages, which are named pieces of code that are tied to a particular object. In this way objects are *communicative*, and groups of objects accomplish tasks by sending messages to each other.
- Objects consist of encapsulated code (methods or operations) along with a mutable state (instance variables or fields). Right away it should be clear that objects are inherently non-functional. This mirrors the statefulness of everyday objects.
- Objects are typically organized in a *nested* fashion. For example, consider an object that represents a graphical web browser. The object itself is frame, but that frame consists of a toolbar and a viewing area. The toolbar is made up of buttons, a location bar, and a menu, while the viewing area is made up of a rendered panel and a scroll bar. This concept is illustrated in Figure 5.1. Object nesting is generally accomplished by the outer object storing its inner objects as fields or instance variables.
- Objects have unique names, or *object references*, to refer to them. This is analogous to the naming of real-world objects, with the advantage that object references are always unique whereas real-world object names may be ambiguous.
- Objects are *self aware*, that is, objects contain references to themselves. `this` in Java and `self` in Smalltalk are self-references.
- Objects are *polymorphic*, meaning that a "fatter" object can always be passed to a method that takes a "thinner" one, that is, one with fewer methods and public fields.

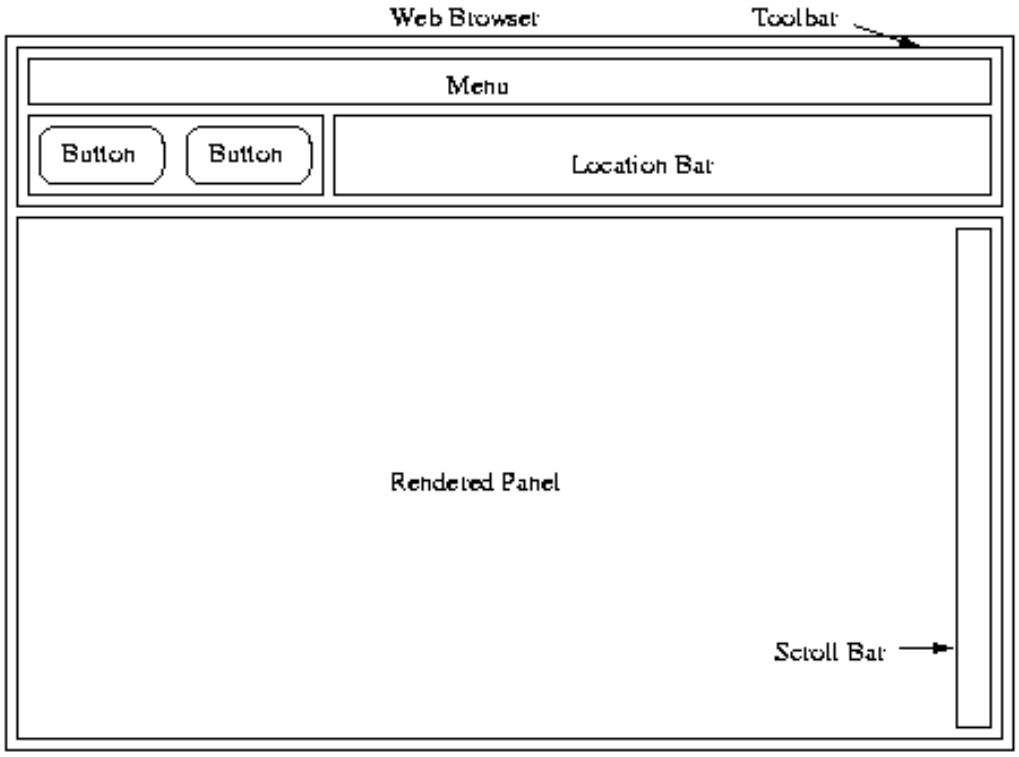


Figure 5.1: A nested object representing a web browser.

There are several additional features objects commonly have. *Classes* are nearly always present in languages with objects. Classes are not required: it is perfectly valid to have objects without classes. The language Self [19, 20, 2] has no classes, instead it has *prototype* objects which are copied that do the duty of classes. JavaScript copied this idea from Self and so now many programmers are using prototype-based objects. Important concepts of classes include creation, inheritance, method overriding, superclass access, and dynamic dispatch. We will address these concepts later.

Information hiding for fields and methods is another feature that most object-oriented languages have, generally in the form of `public`, `private`, and `protected` keywords.

Other aspects of objects include object types and modules. Types are discussed briefly, but modules are beyond the scope of this book. We also ignore method *overloading*, as it is simply syntactic sugar and adds only to readability, not functionality. Recall that overloading a method means there are two methods with the same name, but different type signatures. Overriding means redefining the behavior of a superclass's method in a subclass. Overriding is discussed in Section 5.1.5.

5.1 Encoding Objects in F♭SR

One of the most important aspects of objects is how close they are to existing concepts that we have already discussed. In this section we demonstrate how objects may easily be encoded in **F♭SR**. We will model objects as records of functions and references. Record labels and values can be thought of as slots for either methods or fields. A slot with a function is a method, and a slot with a reference is a field.

5.1.1 Simple Objects

Consider the object in Figure 5.2[2] that represents a point in two-dimensional space. The object has fields `x` and `y`, along with two methods: `magnitude` and `iszero`, with obvious functionality. To encode this object as a record, we are going to need a record that has the following structure.

```
Let point = {
   x = 4;
   y = 3;
   magnitude = Function _ -> ...;
   iszero = Function _ -> ...
} In ...
```

We can't write the `magnitude` method yet, because we need to define it in terms of `x` and `y`, but there is no way to refer to them in the function body. The solution we will use is the same one that C++ uses behind the scenes: we pass the object itself as an argument to the function. Let's revise our first encoding to the following one (assuming we've defined a `sqrt` function).

[2]We use UML to diagram objects. UML is fully described in [11]. Although the diagram in Figure 5.2 is technically a class diagram, for our purposes we will view it simply as the induced object.

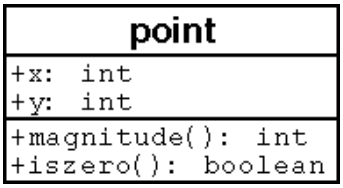


Figure 5.2: The "point" object.

```
Let point = {
   x = 4;
   y = 3;
   magnitude = Function this -> Function _ ->
      sqrt(sqr this.x + sqr this.y);
   iszero = Function this -> Function _ ->
      ((this.magnitude) this {}) = 0
} In ...
```

There are a few points of interest about the above example. First of all, the object itself needs to be explicitly passed as an argument when we send a message. For example, to send the magnitude message to `point`, we would write

```
point.magnitude point {}
```

For convenience, we can use the abbreviation `obj <- method` to represent the message `(obj.method obj)`.

Even inside the object's methods, we need to pass `this` along when we call another method. `iszero` illustrates this point in the way it calls `magnitude`. This encoding of self-reference is called the **self-application encoding**. There are a number of other encodings, and we will treat some of them below.

There is a problem with this encoding, though; our object is still immutable. An object with immutable fields can be a good thing in many cases. For example, windows that can't be resized and sets with fixed members can be implemented with immutable fields. In general, though, we need our objects to support mutable fields. Luckily this problem has an easy solution. Consider the following revision of our `point` encoding, which uses `Ref`s to represent fields.

```
Let point = {
   x = Ref 4;
   y = Ref 3;
   magnitude = Function this -> Function _ ->
      sqrt(sqr !(this.x) + sqr !(this.y));
   iszero = Function this -> Function _ ->
      (this.magnitude this {}) = 0;
   setx = Function this -> Function newx -> this.x := newx;
   sety = Function this -> Function newy -> this.y := newy
} In ...
```

To set `x` to `12`, we can write either `point <- setx 12`, or we can directly change the field with `(point.x) := 12`. To access the field, we now write `!(point.x)`, or we could define a `getx` method to abstract the dereferencing. This strategy gives us a faithful encoding of simple objects. In the next few sections, we will discuss how to encode some of the more advanced features of objects.

5.1.2 Object Polymorphism

Suppose we define the following function.

```
Let tallerThan = Function person1 -> Function person2 ->
   greatereq (person1 <- height) (person2 <- height)
```

If we were to define `person` objects that supported the `height` message, we could pass them as arguments to this function. However, we could also create specialized `person` objects, such as `mother`, `father`, and `child`. As long as these objects still support the `height` message, they are all valid candidates for the `tallerThan` function. Even objects like `dinosaur`s and `building`s can be arguments. The only requirement is that any objects passed to `tallerThan` support the message `height`. This is known as **object polymorphism**.

We already encountered a similar concept when we discussed record polymorphism. Recall that a function

```
Let getheight = Function r -> r.height ...
```

can take any record with a `height` field:

```
... In getheight {radius = 4; height = 4; weight = 44}
```

Object polymorphism is really the same as record polymorphism, evidenced by how we view objects as records. So simply by using the record encoding of objects, we can easily get object polymorphism.

```
Let eqpoint = {
   (* all the code from point above: x, y, magnitude ... *)
   equal = Function this -> Function apoint ->
      !(this.x) = !(apoint.x) And !(this.y) = !(apoint.y)
} In eqpoint <- equal({
   x = Ref 3;
   y = Ref 7;
   (* ... *)
})
```

The object passed to `equal` needs only to define `x` and `y`. Object polymorphism in our embedded language is thus more powerful than in C++ or Java: C++ and Java look at the *type* of the arguments when deciding what is allowed to be passed to a method. Subclasses can always be passed to methods that take the superclass, but nothing else is allowed. In our encoding, which is closer to Smalltalk, any object is allowed to be passed to a function or method as long as it supports the messages needed by the function.

One potential difficulty with object polymorphism is that of **dispatch**: since we don't know the form of the object until runtime, we do not know exactly where the correct methods will be laid out in memory. Thus hashing may be required to look up methods in a compiled object-oriented language. This is exactly the same problem that arises when writing the record-handling code in our **F♭SR** compiler (see Chapter 8), again illustrating the similarities between objects and records.

5.1.3 Information Hiding

Most object-oriented languages allow **information hiding** which is used to protect methods and instances variables from direct outside use. Information hiding is one of the key tools for the encapsulation of object data. In C++ and Java, the `public`, `private`, and `protected` qualifiers are used to control the degree of information hiding for both fields and methods.

For now we will simply encode hiding in **F♭SR**. Note that only public and private data makes sense in **F♭SR** (protected data only makes sense in the context of classes and inheritance, which we have not defined yet). In our encoding, we accomplish hiding by simply making data inaccessible. In real, typed languages, it is the type system itself (and the bytecode verifier in the case of Java) that enforces the privacy of data.

Let's begin with a partial encoding of information hiding.

```
Let pointImpl = (* point from before *) In
Let pointInterface = {
   magnitude = pointImpl.magnitude pointImpl;
   setx = pointImpl.setx pointImpl;
   sety = pointImpl.sety pointImpl
} In ...
```

In this encoding, each method is "preapplied" to `pointImpl`, the full point object, while `pointInterface` contains only public methods and instances. Methods are now invoked simply as

```
pointInterface.setx 5
```

This solution has a flaw, though. Methods that return `this` re-expose the hidden fields and methods of the full object. Consider the following example.

```
Let pointImpl = {
   (* ... *)
   sneaky = Function this -> Function _ -> this
} In Let pointInterface = {
   magnitude = pointImpl.magnitude pointImpl;
   setx = pointImpl.setx pointImpl;
   sety = pointImpl.sety pointImpl;
   sneaky = pointImpl.sneaky pointImpl
} In pointInterface.sneaky {}
```

The `sneaky` method returns the full `pointImpl` object instead of the `pointInterface` version. To remedy this problem, we need to change the encoding so that we don't have to pass `this` every time we invoke a method. Instead, we will give the object a pointer to itself from the start.

```
Let prePoint = Let privateThis = {
   x = Ref 4;
   y = Ref 3
} In Function this -> {
   magnitude = Function _ ->
      sqrt !(privateThis.x) + !(privateThis.y);
   setx = Function newx -> privateThis.x := newx;
   sety = Function newy -> privateThis.y := newy;
   getThis = Function _ -> this
} In Let point = prePoint prePoint In ...
```

Now the message send operation is just

```
point.magnitude {}
```

The method `getThis` still returns `this`, but `this` contains the public parts only. Note that for this encoding to work, `privateThis` can be used only as a target for messages, and can not be returned.

The disadvantage of this encoding is that `this` will need to be applied to itself every time it's used inside the object body. For example, instead of writing

```
(point.getThis {}).magnitude {}
```

we would, instead, have to write

```
((point.getThis {}) (point.getThis {})).magnitude {}
```

These encodings are relatively simple. Classes and inheritance are more difficult to encode, and are discussed below. Object typing is also particularly difficult, and is covered in Chapter 6.

5.1.4 Classes

Classes are foremost templates for creating objects. A class is, in essence, an object factory. Each object that is created from a class must have its own unique set of instance variables (with the exception of static fields, which we will treat later).

It is relatively easy to produce a simple encoding of classes. Simply freeze the code that creates the object, and thaw to create new object. Ignoring information hiding, the encoding looks as follows.

```
Let pointClass = Function _ -> {
   x = Ref 4;
   y = Ref 3;
   magnitude = Function this -> Function _ ->
      sqrt !(this.x) + !(this.y);
   setx = Function this -> Function newx -> this.x := newx
   sety = Function this -> Function newy -> this.y := newy
} In ...
```

We can define `new pointClass` to be `pointClass {}`. Some typical code which creates and uses instances might look as follows.

```
Let point1 = pointClass {} In
Let point2 = pointClass {} In
point1 <- setx 5 ...
```

`point1` and `point2` will have their own `x` and `y` values, since `Ref` creates new store cells each time it is thawed. The same freeze and thaw trick can be applied to our encoding of information hiding to get hiding in classes. The difficult part of encoding classes is encoding inheritance, which we discuss in the next section.

A more useful notion of class is to not just freeze an object, but make a function returning the object; the parameters to that function are analogous to the constructor values of a class.

```
Let pointClass' = Function ix -> Function iy -> {
   x = Ref ix;
   y = Ref iy;
   magnitude = Function this -> Function _ ->
      sqrt !(this.x) + !(this.y);
   setx = Function this -> Function newx -> this.x := newx
   sety = Function this -> Function newy -> this.y := newy
} In ...
```

Then `pointClass' 0 0` creates a point object initialized to $(0, 0)$.

5.1.5 Inheritance

As a rule, about 80 percent of the utility of objects is realized in the concepts we have talked about above, namely

- Objects that encapsulate data and code under a single name to achieve certain functionality.
- Polymorphic objects.
- Templates for generating objects (the main function of classes)

The other 20 percent of the utility comes from from **inheritance**. Inheritance allows related objects to be defined that share some common code. Inheritance can be encoded in **F♭SR** by using the following scheme. In the subclass, we create an instance of the superclass object, and keep the instance around as a "slave." We use the slave to access methods and fields of the superclass object. Thus, the subclass object only contains new and overridden methods. Real object oriented languages tend not to implement inheritance this way for reasons of efficiency (imagine a long inheritance chain in which a method call has to be delegated all the way back to the top of the hierarchy before it can be invoked). Still, the encoding is good enough to illustrate the main points of inheritance. For example, consider the following encoding of `ColorPoint`, a subclass

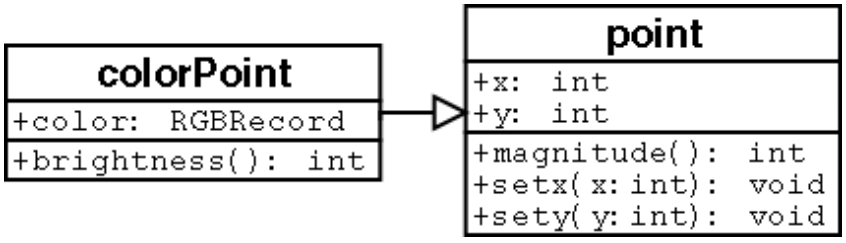


Figure 5.3: The `point`, `colorPoint` inheritance hierarchy.

of `Point`, illustrated in Figure 5.3. (In this encoding we will assume there are no class constructor parameters, i.e. we will work from `pointClass` and not `pointClass'` above.)

```
Let pointClass = ... In
Let colorPointClass = Function _ ->
   Let super = pointClass {} In {
   x = super.x; y = super.y;
   color = Ref {red = 45; green = 20; blue = 20};
   magnitude = Function this -> Function _ ->
      mult(super.magnitude this {})(this.brightness this {});
   brightness = Function this -> Function _ ->
      (* compute brightness... *)
   setx = super.setx; sety = super.sety
} In ...
```

There are several points of interest in this encoding. First of all, notice that to inherit methods and fields from the superclass, we explicitly link them together (i.e. `x`, `y`, `setx`, and `sety`). To override a method in the superclass, we simply redefine it in the subclass instead of linking to the superclass; `magnitude` is an example of this. Also notice that we can still invoke superclass methods from the subclass. For instance, `magnitude` invokes `super.magnitude` in its body. Notice how `super.magnitude` is passed `this` instead of `super` as its argument. This has to do with dynamic dispatch, which we will address now.

5.1.6 Dynamic Dispatch

We say that a method is dynamically dispatched if, looking at a message send v`<-`m, we are not sure precisely what method m will be executed at runtime. Dynamic dispatch is related to object polymorphism: a variable v could contain many different kinds of objects, so v`<-`m could be sending m to any one of those different kinds of objects. Unless we know what kind of objects v is at runtime, we cannot know which method m will be invoked.

The term **dynamic dispatch** refers to how the method invoked by a message send is not fixed at compile-time. If we had a method `isNull` declared in `pointClass` and inherited by `colorPointClass` with code

```
Function this -> Function _ -> (this.magnitude this {}) = 0,
```

the `magnitude` method here is not fixed at compile-time. Concretely, assuming `pointClass` has this `isNull` and we execute

```
Let p = pointClass {} In
Let cp = colorPointClass {} In
p <- isNull {}; cp <- isNull {};
```

In `p`'s call to `isNull`, `this` is a `point`, and so `isNull` will internally invoke `point`'s `magnitude` method. On the other hand for `cp`, `this` is a `colorPoint`, and so `colorPoint`'s `magnitude` method will be invoked. In conclusion, the `magnitude` method call from within `isNull` is dynamically dispatched.

The superclass dispatching above also now can be explained. If `isNull` in `colorPointClass` had additionally been overridden to be

```
Function this -> Function _ -> (super.isNull this {}) = 0 And (* etc *),
```

then tracing through `cp`'s call to `isNull` it would have correctly called the `colorPointClass` magnitude, whereas if we had instead written

```
Function this -> Function _ -> (super.isNull super {}) = 0 And (* etc *),
```

(note change in bold), the `pointClass` magnitude would have been invoked from within `isNull` instead, and the brightness would mistakenly not have been taken into account in the magnitude calculation for a `colorPoint`.

Another example For additional clarification on this issue let us do one more example. Consider the following classes, `rectClass` and its subclass `squareClass`.

```
Let rectClass = Function w -> Function l -> {
  getWidth = Function this -> Function _ -> w;
  getLength = Function this -> Function _ -> l;
  area = Function this -> Function _ ->
    mult ((this.getLength) this {}) ((this.getWidth) this {})
} In

Let squareClass = Function e ->
  Let super = rectClass 1 e In {

  getLength = (super.getLength);
```

```
  (* We override width to be the same as length *)
  getWidth = (super.getLength);

  areaStatic = Function this -> Function _ ->
    (super.area) super {};
  areaDynamic = Function this -> Function _ ->
    (super.area) this {}
} In
Let mySquare = squareClass 10 In
mySquare <-areaDynamic {} (* returns 100 *);
mySquare <-areaStatic {} (* returns 10 *)
```

Notice that in the `squareClass`, `getLength` has been overridden to behave the same as `getWidth`. There are two ways to calculate the area in `squareClass`. `areaStatic` calls `(super.area) super {}`. This means that when `rectClass`'s `area` method is invoked, it is invoked with `rectClass`'s `getLength` and `getWidth` as well. The result is $1 \times 10 = 10$ since the rectangle was initialized to have width 1.

On the contrary, the dynamically dispatched area method, `areaDynamic` is written `(super.area) this {}`. This time, `rectClass`'s `area` method is invoked, but `this` is an instance of `squareClass`, rather than `rectClass`, and `squareClass`'s overridden `getLength` and `getWidth` are used. The result is `1`, the correct area for a square. The behavior of dynamic dispatch is almost always the behavior we want. The key to dynamic dispatch is that inherited methods get a revised notion of `this` when they are inherited. Our encoding promotes dynamic dispatch, because we explicitly pass `this` into our methods.

Java (and almost every other object-oriented language) uses dynamic dispatch to invoke methods by default. In C++, however, unless a method is declared as `virtual`, it will *not* be dynamically dispatched.

5.1.7 Static Fields and Methods

Static fields and methods are found in almost all object-oriented languages, and are simply fields and methods that are not tied to a particular instance of an object, but rather to the class itself.

We can trivially encode static fields and methods by simply making our class definitions records instead of functions. The creation of the object can itself be a static method of the class. This is, in fact, how Smalltalk implements constructors. Consider the following reimplementation of `pointClass` in which we make use of static fields.

```
Let pointClass = {

  newWithXY = Function class ->
    Function newx ->
    Function newy -> {

    x = Ref newx;
    y = Ref newy;
```

```
    magnitude = Function this -> Function _ ->
      sqrt ((!(this.x)) + (!(this.y)))
  };

  new = Function class -> Function _ ->
    (class.newWithXY) class (class.xdefault)
                            (class.ydefault);

  xdefault = 4;
  ydefault = 3
} In

Let point = (pointClass.new) pointClass {} In
(point.magnitude) point {}
```

Notice how the class method `newWithXY` is actually responsible for building the `point` object. `new` simply invokes `newWithXY` with some default values that are stored as class fields. This is a very clean way to encode multiple constructors.

Perhaps the most interesting thing the encoding is how classes with static fields start to look like our original encoding of simple objects. Look closely—notice that class methods take an argument `class` as their first parameter, for the exact same reason that regular methods take `this` as a parameter. So in fact, `pointClass` is really just another primitive object that happens to be able to create objects.

Viewing classes as objects is the dominant paradigm in Smalltalk. Java has some support for looking at classes as objects through the reflection API as well. Even in languages like C++ that don't view classes as objects, design patterns such as the *Factory* patterns[12] capture this notion of objects creating objects.

This encoding is truly in the spirit of object-oriented programming, and it is a clean, and particularly satisfying way to think about classes and static members.

5.2 The F♭OB Language

Now that we have looked at encodings of objects in terms of the known syntax of **F♭SR**, we may now study how to add these features directly to a language. We will call this language **F♭OB**, **F♭** with objects. Our encodings of objects were operationally correct, but not adding syntactic support for objects makes them too difficult to work with in practice. The `colorPoint` encoding, for example, is quite difficult to read. This readability problem only gets worse when types are introduced into the language.

F♭OB includes most of the features we discussed above: classes, message send, methods, fields, `super`, and `this`. We support information hiding in the same manner in which Smalltalk does: all instance variables are hidden (protected) and all methods are exposed (public).

F♭OB also supports **primitive objects**. Primitive objects are objects that are defined "inline," that is, objects that are not created from a class. They are a more lightweight form of object, and are similar to Smalltalk's blocks and Java's anonymous classes. Primitive objects aren't very common in practice, but we include them in **F♭OB** because it requires very little work. The value returned by the expression `new aClass` is an object, which means that an object is a first class expression. As long as our concrete syntax allows us to directly define primitive objects, no additional work is needed in the interpreter.

An interesting consequence of having primitive objects is that we could get rid of functions entirely (not methods). Functions could simply be encoded as methods of primitive objects. For this reason, object-oriented languages that support primitive objects have many of the advantages of higher-order functions.

5.2.1 Concrete Syntax

Let's introduce the **F♭OB** concrete syntax with an example. The following code is an implementation of the `pointClass` / `colorPointClass` hierarchy from before (Figure 5.3). For simplicity this encoding does not include class constructor arguments.

```
Let pointClass =
  Class Extends EmptyClass
    Inst
      x = 0;
      y = 0
    Meth
      magnitude = Function _ -> sqrt(x + y);
      setx = Function newx -> x := newx;
      sety = Function newy -> y := newy

In Let colorPointClass =
  Class Extends pointClass
    Inst
      x = 0;
      y = 0;
      (* A use of a primitive object: *)
      color = Object
        Inst
        Meth red = 0; green = 0; blue = 0
    Meth
      magnitude =
        Function _ ->
          mult(Super <- magnitude {})(This <- brightness {})
      (* An unnormalized brightness metric *)
      brightness = Function _ ->
        color <- red + color <- green + color <- blue;
      setx = Super <- setx (* explicitly inherit *)
```

```
      sety = ...; setcolor = ...

In Let point = New pointClass
In Let colorPoint = New colorPointClass In
  (* Some sample expressions *)
  point <- setx 4;  point <- sety 7;
  point <- magnitude{};
  colorpoint <- magnitude {}
```

There is a lot going on with this syntax, so let's take some time to point out some of the major elements. First of all, notice that `This` and `Super` are special "reserved variables." In our **F♭** encodings, we had to write "`Function this ->` " and pass `this` as an explicit parameter. Now, self-awareness happens implicitly, and is `This` is a reference to self.

Note the use of a primitive object to define the `color` field of the `colorPointClass`. The `red`, `green`, and `blue` values are implemented as methods, since fields are always "private."

In our previous encodings we used one style when defining the base class, and another different style when defining the subclass. In **F♭OB** we use the same syntax for both by always specifying the superclass. To allow for base classes that do not inherit from any other class, we allow a class to extend `EmptyClass`, which is simply a special class that does not define anything.

F♭OB instance variables use the l/r-value form of state that was discussed in Section 4.1.3. There is no need to explicitly use the `!` operator to get the value of an instance variable. **F♭OB** instance variables are therefore mutable, not following the Caml convention that all variables are immutable. Note that method arguments are still immutable. There are thus two varieties of variable: immutable method parameters, and mutable instances. Since it is clear which variables are instances and which are not, there is no great potential for confusion. It is the job of the parser to distinguish between instance variables and regular variables. An advantage of this approach is that it keeps instance variables from being directly manipulated by outsiders.

Method bodies are generally functions, but need not be; they can be any immutable, publicly available value. For example, immutable instances can be considered methods (see the `color` primitive object in the example above).

Note that we still have to explicitly inherit methods. This is not the cleanest syntax, but it simplifies the interpreter, and facilitates the translation to **F♭SR** discussed in Section 5.2.3.

Also, there is no constructor function. `new` is used to create new instances, and, following Smalltalk, initialization is done explicitly by writing an initialize method.

For simplicity, **F♭OB** does not support static fields and methods, nor does it take the "classes as objects" view discussed in the previous section.

5.2.2 A Direct Interpreter

We first consider a direct interpreter for **F♭OB**. The abstract syntax can be expressed by the following Caml type.

```
type ide = Ide of string | This | Super

type label = Lab of string

type expr = (* the Fb expressions, including Let *)
  (* Object holds the instance list and the method list *)
| Object of ((label * expr) list) * ((label * expr) list)
| Class of expr * ((label * expr) list) * ((label * expr) list)
| EmptyClass
| New of expr
| Send of  expr * label
  (* parser has to decide if a var. is InstVar or just a Var *)
| InstVar of label
| InstSet of label * expr
```

Here is a rough sketch of the interpreter. This interpreter is not complete, but it gives a general idea of what one should look like.

```
(* Substitute "sinst" for Super in all method bodies *)
let rec subst_super sinst meth =
  match meth with
    [] -> []
  | (l, body)::rest ->
      (l, subst(body, InstVar(sinst), Super))::
        (subst_super sinst rest)

let rec eval e =
  match e with
    ...
  | Object(inst, meth) -> Object(eval_insts inst, meth)
  | Send(term1, label) ->
      (match (eval term1) with
         Object(inst, meth) ->
           subst(selectMeth(meth, label),
                 Object(inst, meth), This)
         _ -> raise TypeMismatch)
  | Class(super, inst, meth) -> Class(super, inst, meth)
  | New(Class(super, inst, meth)) ->
      (match super with
         EmptyClass -> eval (Object(inst, meth))
       | s -> let sobj = eval(New super) in
              let sinst = (* A fresh instance variable label *) in
              let newinst = (sinst, sobj)::inst in
              let newmeth = subst_super sinst meth in
              eval (Object(newinst, newmeth))
  ...
```

```
and eval_insts inst =
  match inst with
    [] -> []
  | (l, body)::rest -> (l, eval(body))::(eval_insts rest)
```

This code sketch for the interpreter does a nice job of illustrating the roles of `This` and `Super`. We only substitute for `this` *when we send a message*. That's because `This` is a dynamic construct, and we don't know what it will be until runtime (see the discussion of dynamic dispatching in Section 5.1.6). On the other hand, `Super` is a static construct, and is known at the time we write the code, which allows us to substitute for `Super` as soon as the `Class` expression is evaluated.

5.2.3 Translating F♭OB to F♭SR

Another way to give meaning to **F♭OB** programs is by defining a translation mapping **F♭OB** programs into **F♭SR** programs. This is a complete characterization of how objects can be encoded in **F♭SR**, the topic of Section 5.1.

In Chapter 8 below, we devleop a compiler for **F♭SR**. To obtain a compiler for **F♭OB**, we can simply add a translation step that translates **F♭OB** to **F♭SR**, and then simply compile the **F♭SR** using this compiler. Thus, we will have a **F♭OB** compiler "for free" when we are all done. This section only discusses the **F♭OB** to **F♭SR** translation.

The translation is simply a formalization of the encodings given in Section 5.1. Although real object-oriented compilers are much more sophisticated, this section should at least provide an understandable view of what an object-oriented compiler does. The concrete syntax translation is inductively defined below in a piecewise manner.

$toFbSR$(`Object Inst` x_1`=`e_1`;` $\ldots$`;` x_n`=`e_n `Meth` m_1`=`e'_1`;` $\ldots$`;` m_k`=`e'_k) =
  {`inst =` {x_1`=Ref(`$toFbSR(e_1)$`);` $\ldots$`;` x_n`=Ref(`$toFbSR(e_n)$`)};
   `meth =` {m_1`= Function this ->` $toFbSR(e'_1)$`;` $\ldots$`;`
    m_k`= Function this ->` $toFbSR(e'_k)$`}` }
$toFbSR$(`Class Extends` e `Inst` x_1`=`e_1`;` $\ldots$`;` x_n`=`e_n
  `Meth` m_1`=`e'_1`;` $\ldots$`;` m_k`=`e'_k) =
  `Function _ -> Let super =` ($toFbSR(e)$) {} `In`
    {`inst =` {x_1`=Ref(`$toFbSR(e_1)$`);` $\ldots$`;` x_n`=Ref(`$toFbSR(e_n)$`)};
     `meth =` {m_1`= Function this ->` $toFbSR(e'_1)$`;` $\ldots$`;`
      m_k`= Function this ->` $toFbSR(e'_k)$`}` }
$toFbSR$(`New` e) = ($toFbSR(e)$) {}
$toFbSR$(`EmptyClass`) = `Function _ ->` {}
$toFbSR$(`Super <-` m *args*) = `super.meth.`m `this` *args*
$toFbSR$(e `<-` m *args*) =
  `Let ob =` $toFbSR(e)$ `In ob.meth.`m `ob` *args*, for e not `Super`
$toFbSR$(x `:=` e) = `this.inst.`x `:=` $toFbSR(e)$
$toFbSR$(x) = `!(this.inst.`x`)`, for x an instance variable
$toFbSR$(y) = y, for y a function variable
$toFbSR$(anything else) = homomorphic

The translation is fairly clean, except that messages to `Super` have to be handled a bit differently than other messages in order to properly implement dynamic dispatch. Notice again that instance variables are handled differently than function variables, because they are mutable. Empty function application, i.e. `f ()`, may be written as empty record application: `f {}`. The "`_`" variable in `Function _ ->` e is any variable not occurring in e. The character "`_`" itself is a valid variable identifier in the FbDK implementation of **F♭SR**, however (see Appendix **??**).

As an example of how this translation works, let us perform it on the **F♭OB** version of the `point` / `colorPoint` classes from Section 5.2.1. The result is the following:

```
Let pointClass =
  Function _ -> Let super = (Function _ -> {}) {} In {
    inst = {
      x = Ref 3;
      y = Ref 4
    };
    meth = {
      magnitude = Function this -> Function _ ->
        sqrt ((!(this.inst.x)) + (!(this.inst.y)));
      setx = Function this -> Function newx ->
        (this.inst.x) := newx;
      sety = Function this -> Function newy ->
        (this.inst.y) := newy
    }
}

In Let colorPointClass =
  Function _ -> Let super = pointClass {} In {
    inst = {
      x = Ref 3;
      y = Ref 4;
      color = Ref ({inst = {}; meth = {
        red = Function this -> 45;
        green = Function this -> 20;
        blue = Function this -> 20
      }})
    };
    meth = {
      magnitude = Function this -> Function _ ->
        mult ((super.meth.magnitude) this {})
             ((this.meth.brightness) this {});
      brightness = Function this -> Function _ ->
        (((!(this.inst.color)).meth.red) this) +
        (((!(this.inst.color)).meth.green) this) +
        (((!(this.inst.color)).meth.blue) this);
      setx = Function this -> Function newy ->
```

```
        (super.meth.setx) this newy;
      sety = Function this -> Function newy ->
        (super.meth.setx) this newy;
      setcolor = Function this -> Function c ->
        (this.inst.color) := c
    }
} In

(* Let colorPoint = New colorPointClass In
 *   colorPoint <- magnitude {}
 *)
Let colorPoint = colorPointClass {} In
  (colorPoint.meth.magnitude) colorPoint {};;
```

Interact with F♭SR. The translated code above should run fine in **F♭SR**, provided you define the `mult` and `sqrt` functions first. `mult` is easily defined as

```
Let Rec mult x = Function y ->
  If y = 0 Then
    0
  Else
    x + (mult x (y - 1)) In ...
```

`sqrt` is not so easily defined. Since we're more concerned with the behavior of the objects, rather than numerical accuracy, just write a dummy `sqrt` function that returns its argument:

```
Let sqrt = Function x -> x In ...
```

Now, try running it with the **F♭SR** file-based interpreter. Our dummy `sqrt` function returns `7` for the `point` version of magnitude, and the `colorPoint` magnitude multiplies that result by the sum of the brightness (`85` in this case). The result is

```
$ FbSR fbobFBsr.fbsr
==> 595
```

After writing a Caml version of *toFbSR* for the abstract syntax, a **F♭OB** compiler is trivially obtained by combining *toFbSR* with the functions defined in Chapter 8:

```
let FbOBcompile e = toC(hoist(atrans(clconv(toFbSR e))))
```

Finally, there are several other ways to handle these kinds of encodings. More information about encoding objects can be found in [10].

Chapter 6

Type Systems

In **F♭**, if we evaluate the expression

```
3 + (If False Then 3 Else False),
```

we will get some kind of interpreter-specific error at runtime. If **F♭** had a type system, such an expression would not have been allowed to evaluate.

In Lisp, if we define a function

```
(defun f (x) (+ x 1)),
```

and then call `(f "abc")`, the result is a runtime type error. Similarly, the Smalltalk expression

```
String new myMessage
```

results in a "message not supported" exception when run. Both of these runtime errors could have been detected before runtime if the languages supported static type systems.

The C++ code

```
int a[10]; a[123] = 5;
```

executes unknown and potentially harmful effects, but the equivalent Java code will throw an `ArrayIndexOutOfBoundsException` at runtime, because array access is checked by a dynamic type system.

These are just a few examples of the kinds of problems that type systems are designed to address. In this chapter we discuss such type systems, as well as algorithms to infer and to check types.

6.1 An Overview of Types

A **type** is simply a property with which a program is implicitly or explicitly annotated before runtime. Type declarations are invariants that hold for *all* executions of a program, and can be expressed as statements such as "this variable always holds a `String` object," or "this function always returns a `tree` expression."

Types have many other advantages besides simply cutting down on runtime errors. Since types (and module signatures) specify invariant properties of the program, they serve as precise and descriptive comments on the functionality of the code. In this manner, types aid in large software development.

With a typed language, more information is known at compile-time, and this helps the compiler produce much faster code. For example, the record implementation we will use in our **F♭SR** compiler (via hashing, see Chapter 8) is not needed in C++ due to its static type system. We know the size of all records at compile-time, and therefore know where they should be laid out in memory. In contrast, Smalltalk is very slow, and the lack of a static type system counts for much of the slowness. The Strongtalk system [8] attempts to bring a static type system into Smalltalk.

Finally, in an untyped language its easier to do very ugly "hacking". For instance, a list `[1;true;2;false;3;true]` can be written in an untyped language, but this is dangerous and it would be much preferred to represent this as `[(1,true);(2,false);(3,true)]`, which, in OCaml, has the type `(int * bool) list`.

However, untyped languages have one distinct advantage over typed ones: they are more expressive. For example, consider the **F♭SR** encoding of **F♭OB** from Chapter 5. This encoding would not work with OCaml as the target language, because the OCaml type system disallows record polymorphism. The Y-combinator is another example of where an untyped language really shines. **F♭** could support recursion via the Y-combinator, but a simple typed version of **F♭** can't since it cannot be typed. Recall from Section 2.3.5 that OCaml can not type the Y-combinator, either.

All of us have seen types used in many ways. The following is a list of some of the more common dimensions of types.

- Atomic types: `int`, `float`, ...
- Type constructors, which produce types from types: `'a -> 'b`, `'a * 'b`
- OCaml-style type constructor definitions via `type`
- C-style type definitions via `struct` and `typedef`
- Object-oriented types: class types, object types
- Module types, or signatures
- Java-style interfaces
- Exception types: ⟨*method*⟩ `throws` ⟨*exception*⟩
- Parametric polymorphism: `'a -> 'a`

- Record/Object polymorphism: pass a `ColorPoint` to a function which expects a `Point`.

There are also several newer dimensions of types that are currently active research areas.

- Effect types: the type "`int -x,y-> int`" indicates that variables `x` and `y` will assigned to in this function. Java's `throws` clauses for methods are a form of effect types.
- Concrete class analysis: for variable `x:Point`, a concrete class analysis produces a set such as {`Point`, `ColorPoint`, `DataPoint`}. This means at runtime `x` could either be a `Point`, a `ColorPoint`, or a `DataPoint` (and, nothing else). This is useful in optimization.
- Typed Assembly Language [3, 17]: put types on assembly-level code and have a type system that guarantees no unsafe pointer operations.
- Logical assertions in types: `int -> { x:int | odd(x) }` for a function returning odd numbers.

There is an important distinction that needs to be made between *static* and *dynamic* type systems. **Static type systems** are what we usually mean when we talk about type systems. A static type system is the standard notion of type found in C, C++, Java and OCaml. Types are checked by the compiler, and type-unsafe programs fail to compile.

Dynamic type systems, on the other hand, check type information at runtime. Lisp, Scheme, and Smalltalk are, in fact, dynamically typed. In fact, **F♭** and **F♭SR** are technically dynamically typed as well, since they will raise a `typeMismatch` when the type of an expression is not what it expects. Any time you use a function, the runtime environment makes sure that its a function. If you use an integer, it makes sure it's an integer, etc. These runtime type checks add a lot of overhead to the runtime environment, and thus cause programs to run slowly.

There is some dynamic typechecking that occurs in statically typed languages too. For instance, in Java, downcasts are verified at run-time and can raise exceptions. Out-of-bounds array accesses are also checked at run-time in Java and OCaml, and are thus dynamically typed. Array accesses are not typed in C or C++, since no check is performed at all. Note that the *type* of an array (i.e. `int`, `float`) is statically checked, but the *size* is dynamically checked.

Finally, languages can be **untyped**. The FbSR *compiler* produces untyped code, since runtime errors cause core dumps. It is important to understand the distinction between an *untyped* language and a *dynamically typed* one. In an untyped language there is no check at all and anomalous behavior can result at runtime. In Chapter 8, we compile **F♭SR** to untyped C code, using casts to "disable" type system. Machine language is another example of an untyped language.

To really see the difference between untyped and dynamically typed languages, consider the following two program fragments. The first is C++ code with the type system "disabled" via casts.

```
#include <iostream>

class Calculation {
 public: virtual int f(int x) { return x; }
};

class Person {
 public: virtual char *getName() { return "Mike"; }
};

int main(int argc, char **argv) {
  void *o = new Calculation();
  cout << ((Person *)o)->getName() << endl;

  return 0;
}
```

The code compiles with no errors, but when we run it the output is "ã¿." But if we compile it with optimization, the result is "Ãà." Run it on a different computer and it may result in a segmentation fault. Use a different compiler, and the results may be completely exotic and unpredictable. The point is that because we're working with an untyped language, there are no dynamic checks in place, and meaningless code like this results in undefined behavior.

Contrast this behavior with that of the equivalent piece of Java code. Recall that Java's dynamic type system checks the type of the object when a cast is made. We will use approximately the same code:

```
class Calculation {
  public int f(int x) { return x; }
}

class Person {
  public String getName() { return "Mike"; }
}

class Main {
  public static void main(String[] args) {
    Object o = new Calculation();
    System.out.println(((Person) o).getName());
  }
}
```

When we run this Java code, the behavior is quite predictable.

```
Exception in thread "main" java.lang.ClassCastException: Calculation
        at Main.main(example.java:12)
```

The `ClassCastException` is an exception raised by Java's dynamic type system. The unsafe code is never executed in Java, and so the behavior of the program is consistent and well-defined. With the C++ version, there is no dynamic check, and the unsafe code *is* executed, resulting in wild and unexpected behavior.

6.2 TF♭: A Typed F♭ Variation

We will use the prefix "**T**" to represent a typed version of a language which we have previously studied. Thus we have several possible languages: **TF♭**, **TF♭R**, **TF♭S**, **TF♭SR**, **TF♭OB**, **TF♭X**, **TF♭SRX**, etc. There are too many languages to consider individually, so we will first look at **TF♭** as a warm-up, and then consider full-blown **TF♭SRX**.

6.2.1 Design Issues

Before we begin to investigate **TF♭** or any typed language, there are a few general design issues to address. We will take a moment to discuss these issues before giving the specification for **TF♭**.

The first question to ask is how much explicit type information must our language contain? How much type information must the program be decorated with, and how much can be inferred by the compiler? A spectrum of possibilities exists.

On one end of the spectrum, we can use no decoration at all. We simply stick to our untyped language syntax, and let the compiler *infer* all the type information.

Alternatively, we can use limited decoration, and have the compiler do partial inference. There is a wide range of possibilities. C for instance requires function argument and return types to be specified, and declared variables to be given types. However, within the body of a function, individual expressions do not need to be typed as those types may be inferred. Some languages only require function argument types be declared, and the return values of functions is then inferred.

At the other end of the spectrum, every subexpression and its identifier must be decorated with its type. This is too extreme, however, and makes the language unusable. Instead of writing

```
Function x -> x + 1,
```

we would need some gross syntax like

```
(Function x -> (x:int + 1:int):int):(int -> int).
```

For **TF♭** and **TF♭SRX**, we will concentrate on the C and Pascal view of explicit type information. We specify function argument and return types, and declared variable types, and allow the rest to be inferred.

We should also illustrate the difference between type checking and type inference. In any typed language, the compiler should **typecheck** the program before generating

code. **Type inference algorithms** infer types *and* check that the program body has no type errors. OCaml is an example of this.

A **type checker** generally just checks the body is well-typed given the types listed on declarations. This is how C, C++, and Java work, although technically they are also inferring some types, such as the type of `3+4`.

In any case it must be possible to run the type inference or type checking algorithm quickly. OCaml in theory can take exponential time to infer types, but in practice it is linear.

6.2.2 The TF♭ Language

Finally, we are ready to look at **TF♭**, a typed **F♭** language. To simplify things, we will not include the `Let Rec` syntax of **F♭**, but only non-recursive, anonymous functions. We will discuss typing recursion in Section 6.4.

In analogue with our development of operational semantics and interpreters, we will define two things when discussing types. First we define **type systems**, which are language-independent notations for assigning types to programs. Type systems are analogous to operational semantics. Secondly, we define **type checkers**, which are OCaml implementations of type systems analogous to interpreters. We begin with type systems.

Type Systems

Type systems are rule-based formal systems that are similar to operational semantics. Type systems rigorously and formally specify what program have what types, and have a strong and deep parallel with formal logic (recall our discussion of Russell's Paradox in Section 2.3.5). Type systems are generally a set of rules about type assertions.

Definition 6.1. *(Type Environment) A* ***type environment****,* Γ*, is a set* $\{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ *of bindings of free variables types. If a variable* x *is listed twice in* Γ*, the rightmost (innermost) binding is the proper type. We write* $\Gamma(x) = \tau$ *to indicate that* τ *is the innermost type for* x *in* Γ*.*

Definition 6.2. *(Type Assertion) A* ***type assertion****,* $\Gamma \vdash e : \tau$*, indicates that in type environment* Γ*,* e *is of type* τ*.*

The **TF♭** types in the concrete syntax are

τ `::= Int | Bool |` τ `->` τ.

We can represent this in a OCaml abstract syntax as

```
type fbtype = Int | Bool | Arrow of fbtype * fbtype
```

The expressions of **TF♭** are almost identical to those of **F♭**, except that we must explicitly decorate functions with type information about the argument. For example, in the concrete syntax we write

`Function` x`:`τ `->` e

where the abstract syntax representation is

```
Function of ide * fbtype * expr
```

The TF♭ Type Rules

We are now ready to define the **TF♭** types rules. These rules have the same structure as the operational semantics rules we have looked at before; the horizontal line reads "implies." The following three rules are the axioms for out type system (recall that an axiom is a rule that is always true, that is, a rule with nothing above the line).

(Hypothesis) $$\frac{}{\Gamma \vdash x : \tau \text{ for } \Gamma(x) = \tau}$$

(`Int`) $$\frac{}{\Gamma \vdash n : \texttt{Int} \text{ for } n \text{ an integer}}$$

(`Bool`) $$\frac{}{\Gamma \vdash b : \texttt{Bool} \text{ for } b \texttt{ True} \text{ or } \texttt{False}}$$

The *Hypothesis* rule simply says that if a variable x contained in a type environment Γ has type τ then the assertion $\Gamma \vdash x : \tau$ is true. The *`Int`* and *`Bool`* rules simply give types to literal expressions such as `7` and `False`. These rules make up the base cases of our type system.

Next, we have rules for simple expressions.

(+) $$\frac{\Gamma \vdash e : \texttt{Int}, \quad \Gamma \vdash e' : \texttt{Int}}{\Gamma \vdash e \texttt{ + } e' : \texttt{Int}}$$

(-) $$\frac{\Gamma \vdash e : \texttt{Int}, \quad \Gamma \vdash e' : \texttt{Int}}{\Gamma \vdash e \texttt{ - } e' : \texttt{Int}}$$

(=) $$\frac{\Gamma \vdash e : \texttt{Int}, \quad \Gamma \vdash e' : \texttt{Int}}{\Gamma \vdash e \texttt{ = } e' : \texttt{Bool}}$$

These rules are fairly straightforward. For addition and subtraction, the operands must typecheck to `Int`s, and the result of the expression is an `Int`. Equality is similar, but typechecks to a `Bool`. Note that equality will only typecheck with integer operands, not boolean ones. The *`And`*, *`Or`*, and *`Not`* rules are similar, and their definition should be obvious.

The *If* rule is a bit more complicated. Clearly, the conditional part of the expression must typecheck to a `Bool`. But what about the `Then` and `Else` clauses? Consider the following expression.

`If` e `Then 3 Else False`

Should this expression typecheck? If e evaluates to `True`, then the result is `3`, an `Int`. If e is `False`, the result is `False`, a `Bool`. Clearly, then this expression should not typecheck, because it does not always evaluate to the same type. This tells us that for an `If` statement to typecheck, both clauses must typecheck to the same type, and the type of the entire expression is then the same as the two clauses. The rule is as follows.

$$\textit{(If)} \quad \frac{\Gamma \vdash e : \texttt{Bool}, \quad \Gamma \vdash e' : \tau, \quad \Gamma \vdash e'' : \tau,}{\Gamma \vdash \texttt{If } e \texttt{ Then } e' \texttt{ Else } e'' : \tau}$$

We have now covered all the important rules except for functions and application. The *Function* rule is a bit different from other rules, because functions introduce new variables. The type of the function body depends on the type of the variable itself, and will not typecheck unless that variable is in Γ, the type environment. To represent this in our rule, we need to perform the type assertion with the function's variable appended to the type environment. We do this in the following way.

$$\textit{(Function)} \quad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash (\texttt{Function } x{:}\tau \texttt{ -> } e) : \tau \texttt{ -> } \tau'}$$

Notice the use of the type constructor `->` to represent the type of the entire function expression. This type constructor should be familiar, as the OCaml type system uses the same notation. In addition, the *Function* rule includes the addition of an assumption to Γ. We assume the function argument, x, is of type τ, and add this assumption to the environment Γ to derive the type of e. The *Application* rule follows from the *Function* rule:

$$\textit{(Application)} \quad \frac{\Gamma \vdash e : \tau \texttt{ -> } \tau', \quad \Gamma \vdash e' : \tau}{\Gamma \vdash e\ e' : \tau'}$$

Just as in operational semantics, a *derivation* of $\Gamma \vdash e : \tau$ is a tree of rule applications where the leaves are axioms (*Hypothesis*, *Int* or *Bool* rules) and the root is $\Gamma \vdash e : \tau$.

Let's try an example derivation.

$\vdash$ `(Function x:Int -> (Function y:Bool ->`
`If y Then x Else x+1)) :`
`Int -> Bool -> Int`
Because by the function rule, it suffices to prove
`x : Int` $\vdash$ `(Function y: Bool -> (If y Then x Else x+1)) :`
`Bool->Int`
Because by the function rule again, it suffices to prove
`x : Int, y : Bool` $\vdash$ `If y Then x Else x+1 : Int`
Because by the *If* rule, it suffices to prove
`x : Int, y : Bool` $\vdash$ `y : Bool`
`x : Int, y : Bool` $\vdash$ `x : Int`
`x : Int, y : Bool` $\vdash$ `x+1 : Int`
all of which either follow by the *Hypothesis* rule or `+` and *Hypothesis*.

Given the above and letting

```
f = (Function x:Int -> (Function y:Bool ->
   If y Then x Else x+1))
```

we then have

$\vdash$ `f 5 True : Int`
Because by the application rule,
$\vdash$ `f : Int -> Bool -> Int`
(which we derived above)
$\vdash$ `5 : Int` by the *Int* rule
And thus
$\vdash$ `f 5 : Bool -> Int` by the *Application* rule.
Given this and
$\vdash$ `True : Bool` by the *Bool* rule
we can get
$\vdash$ `f 5 True : Int` by the *Application* rule.

As we mentioned before, **TF**$\flat$ is a very weak language. No recursive functions can be defined. In fact, all programs are guaranteed to halt. **TF**$\flat$ is thus *normalizing*. Without recursion, **TF**$\flat$ is not very useful. Later we will add recursion to **TF**$\flat$**SRX** and show how to type it.

Exercise 6.1. Try to type the Y-combinator in **TF**$\flat$.

Now that we have a type system for the **TF**$\flat$ language, we can detect whether or not our programs are *well-typed*. But what does this mean? The answer comes in the form of the following **type soundness** theorem.

Theorem 6.1. *If $\vdash e : \tau$, then in the process of evaluating e, a "stuck state" is never reached.*

We will not precisely define the concept of a stuck state. It is basically a point at which evaluation can not continue, such as `0 (Function x -> x)` or `(Function x -> x) + 4`. In terms of a **F♭** interpreter, stuck stated are the cases that raise exceptions. This theorem asserts that a type system prevents runtime errors from occurring. Similar theorems are the goal of most type systems.

6.3 Type Checking

To write an *interpreter* for **TF♭**, we simply modify the **F♭** interpreter to ignore type information at runtime. What we are really interested in is writing a **type checker**. This is the type system equivalent of an interpreter; given the language-independent type rules, define a type checking algorithm in a particular language, namely OCaml.

A type-checking algorithm `typeCheck` takes as input a type environment Γ and an expression e. It either returns the type τ of e or it raises an exception indicating that e is not well-typed in the environment Γ.

Some type systems do not have an easy corresponding type-checking algorithm. In **TF♭** we are fortunate in that the type checker mirrors the type rules in a nearly direct fashion. As was the case with the interpreters, the outermost structure of the expression dictates the rule that applies. The flow of the recursion is that we pass the environment Γ and expression e *down*, and return the result type τ back *up*.

Here is a first pass at the **TF♭** typechecker, `typecheck : envt -> expr -> fbtype`. Γ can be implemented as a `(ident * fbtype)` list, with the most recent item at the front of the list.

```
let rec typecheck gamma e =
  match e with
    (* lookup returns the first mapping of x in gamma *)
    Var x -> lookup gamma x
  | Function(Ide x,t,e1) ->
      let t' = typecheck (((Ide x),t)::gamma) e1 in
      Arrow(t,t')
  | Appl(e1,e2) ->
      let Arrow(t1,t2) = typecheck gamma e1 in
      if typecheck gamma e2 = t1 then
        t2
      else
        raise TypeError
  | Plus(e1,e2) ->
      if typecheck gamma e1 = Int and
         typecheck gamma e2 = Int then
        Int
      else
        raise TypeError
```

```
  | (* ... *)
```

Lemma 6.1. *`typecheck` faithfully implements the **TF♭** type system. That is,*

- $\vdash e : \tau$ *if and only if* `typecheck [] e` *returns* τ*, and*
- `typecheck [] e` *raises a* `typeError` *exception if and only if* $\vdash e : \tau$ *is not provable for any* τ*.*

Proof. Omitted (by case analysis). □

This Lemma implies the `typecheck` function is a sound implementation of the type system for **TF♭**.

6.4 Types for an Advanced Language: TF♭SRX

Now that we've looked at a simple type system and type checker, let us move on to a type system for a more complicated language: **TF♭SRX**. We include just about every piece of syntax we have used up to now, except for **F♭OB**'s classes and objects and **F♭V**'s variants. Here is the abstract syntax defined in terms of a OCaml type.

```
type exnid = string
and expr =
    Var of ident
  | Function of ident * fbtype * expr
  | Letrec of ident * ident * fbtype * expr * fbtype * expr
  | Appl of expr * expr
  | Plus of expr * expr | Minus of expr * expr
  | Equal of expr * expr | And of expr * expr
  | Or of expr * expr | Not of expr
  | If of expr * expr * expr | Int of int | Bool of bool
  | Ref of expr | Set of expr * expr | Get of expr
  | Cell of int | Record of (label * expr) list
  | Select of  label * expr | Raise of expr * fbtype |
  | Try of expr * exnid * ident * expr
  | Exn of exnid * expr

and fbtype =
    Int | Bool | Arrow of fbtype * fbtype
  | Rec of label * fbtype list | Rf of fbtype
```

Next, we will define the type rules for **TF♭SRX**. All of the **TF♭** type rules apply, and so we can move directly to the more interesting rules. Let's begin by tackling recursion. What we're really typing is the `In` clause, but we need to ensure that the rest of the expression is also well-typed.

$$\textit{(Let Rec)} \qquad \frac{\Gamma, f : \tau \texttt{ -> } \tau', x : \tau \vdash e : \tau', \qquad \Gamma, f : \tau \texttt{ -> } \tau' \vdash e' : \tau''}{\Gamma \vdash (\texttt{Let Rec } f\ x{:}\tau \texttt{ = } e{:}\tau' \texttt{ In } e') : \tau''}$$

Next, we move on to records and projection. The type of a record is simply a map of the field names to the types of the values associated with each field. Projection is typed as the type of the value of the field projected. The rules are

$$\textit{(Record)} \quad \frac{\Gamma \vdash e_1 : \tau_1, \ldots, \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \{l_1 \texttt{ = } e_1 \texttt{; } \ldots \texttt{; } l_n \texttt{ = } e_n\} : \{l_1 : \tau_1 \texttt{; } \ldots \texttt{; } l_n : \tau_n\}}$$

$$\textit{(Projection)} \quad \frac{\Gamma \vdash e : \{l_1 : \tau_1 \texttt{; } \ldots \texttt{; } l_n : \tau_n\}}{\Gamma \vdash e.l_i : \tau_i \text{ for } 1 \leq i \leq n}$$

We'll also need to be able to type side effects. We can type `Ref` expressions with the special type τ `Ref`. `Set` and `Get` expressions easily follow.

$$\textit{(Ref)} \quad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{Ref } e : \tau \texttt{ Ref}}$$

$$\textit{(Set)} \quad \frac{\Gamma \vdash e : \tau \texttt{ Ref}, \quad \Gamma \vdash e' : \tau}{\Gamma \vdash e \texttt{ := } e' : \tau}$$

$$\textit{(Get)} \quad \frac{\Gamma \vdash e : \tau \texttt{ Ref}}{\Gamma \vdash \texttt{!}e : \tau}$$

Finally, the other kind of side effects we need to type are exceptions. To type an exception itself, we will simply use the type `Exn`. This allows all exceptions to be typed in the same way. For example, the code

```
If b Then (#IntExn 1) Else (#BoolExn False)
```

will typecheck, and has type `Exn`. We do this to allow maximum flexibility.

Because they alter the flow of evaluation, `Raise` expressions should always typecheck, provided the argument typechecks to an `Exn` type. It is difficult to know what type to give a raise expression, though. Consider the following example.

```
If b Then Raise (#Exn True) Else 4
```

This expression should typecheck to type `Int`. From our `If` rule, however, we know that the `Then` and `Else` clause must have the same type. We infer, therefore, that the `Raise` expression must have type `Int` for the `If` to typecheck. In Section 6.6.2 we see how to handle this inference automatically. For now, we will simply type `Raise` expressions with the arbitrary type τ. Note that this is a perfectly valid thing for a type rule to do, but it is difficult to implement in an actual typechecker.

Next, notice that the `With` clause of the `Try` expression is very much like a function. Just as we did with functions, we will need to decorate the identifier with type information as well. However, as we see below, this decoration can be combined with the final kind of type decoration, which we will discuss now.

Consider the expression

```
Try Raise (#Ex 5)
With #Ex x:Int -> x + 1
```

The type of this expression is clearly `Int`. But suppose the example were modified a bit.

```
Try Raise (#Ex False)
With #Ex x:Int -> x + 1
```

This expression will also type to `Int`. But suppose we were to evaluate the expression using our operational semantics for exceptions. When the exception is raised, `False` will be substituted for `x`, which could cause a runtime type error. The problem is that our operational semantics is ignorant of the type of the exception argument.

We can solve this problem without changing the operational semantics, however. Suppose, instead of writing `#Ex False`, we wrote `#Ex@Bool False`. The `@Bool` would be used by the type rules to verify that the argument is indeed a `Bool`, and the interpreter will simply see the string "`#Ex@Bool`", which will be used to match with the `With` clause. This also eliminates the need for type decoration on the `With` clause identifier, since it serves the same purpose. In a sense, this is very much like overloading a method in Java or C++. When a method is overloaded, the type *and* the method name are needed to uniquely identify the correct method. Our final exception syntax looks like this:

```
Try Raise (#Ex@Bool False)
With #Ex@Int x -> x + 1
```

This expression typechecks and has type `Int`. When evaluated, the result is `Raise #Ex@Bool False`, i.e. the exception is not caught by the `With` clause. This is the behavior we want.

Now that we've got a type-friendly syntax worked out, let's move on to the actual type rules. They are fairly straightforward.

$$\textit{(Raise)} \quad \frac{\Gamma \vdash e : \tau'}{\Gamma \vdash \texttt{(Raise \#}xn\texttt{@}\tau'\ e\texttt{)} : \tau \text{ for arbitrary } \tau}$$

$$\textit{(Try)} \quad \frac{\Gamma \vdash e : \tau, \quad \Gamma, x : \tau' \vdash e' : \tau}{\Gamma \vdash (\texttt{Try}\ e\ \texttt{With \#}xn\texttt{@}\tau'\ x\ \texttt{->}\ e') : \tau}$$

Using these rules, let's type the expression from above:

$\vdash$ (`Try Raise (#Ex@Bool False) With #Ex@Int x -> x + 1`) : `Int`
  Because by the *`Raise`* rule,
  $\vdash$ `Raise (#Ex@Bool False)` : τ for arbitrary τ
    Because by the *`Bool`* rule, $\vdash$ `False` : `Bool`
  And, by the `+` rule
  `x` : `Int` $\vdash$ `x + 1` : `Int`
    By application of the *`Int`* and *Hypothesis* rules

Therefore, by the *`Try`* rule, we deduce the type `Int` for the original expression.

Exercise 6.2. Why are there no type rules for cells?

Exercise 6.3. How else could we support recursive functions in **TF♭SRX** without using `Let Rec`, but still requiring that recursive functions properly typecheck? Prove that your solution typechecks.

Exercise 6.4. Attempt to type some of the untyped programs we have studied up to now, for example, the Y-combinator, `Let`, sequencing abbreviations, a recursive factorial function, and the encoding of lists. Are there any that can not typecheck at all?

Exercise 6.5. Give an example of a non-recursive **F♭SR** expression that evaluates properly to a value, but does not typecheck when written in **TF♭SRX**.

6.5 Subtyping

The type systems that we covered above are reasonably adequate, but there are still many types of programs that have no runtime errors that will nonetheless not typecheck. The first extension to our standard type systems that we would like to consider is what is known as **subtyping**. The main strength of subtyping is that it allows record and object polymorphism to typecheck.

Subtypes should already be a familiar concept from Java and C++. Subclasses are subtypes, and extending or implementing an interface gives a subtype.

6.5.1 Motivation

Let us motivate subtypes with an example. Consider a function

```
Function x:{l:Int} -> (x.l + 1):Int.
```

This function takes as an argument a record with field `l` of type `Int`. In the untyped **F♭R** language the record passed into the function could also include other fields besides `l`, and the call

```
(Function x -> x.l + 1) {l = 4; m = 6}
```

would generate no run-time errors. However, this would not type-check by our **TF♭SRX** rules: the function argument type is different from the type of the value passed in.

The solution is to re-consider record types such as {`m:Int; n:Int`} to mean a record with at least the `m` and `n` fields of type `Int`, but possibly other fields as well, of unknown type. Think about the previous record operations and their types: under this interpretation of record typing, the *Record* and *Projection* rules both still make sense. The old rules are still sound, but we need a new rule to reflect this new understanding of record types:

$$\textit{(Sub-Record}_0\textit{)} \qquad \frac{\Gamma \vdash e : \{l_1 : \tau_1;\ \ldots;\ l_n : \tau_n\}}{\Gamma \vdash e : \{l_1 : \tau_1;\ \ldots;\ l_m : \tau_m\} \text{ for } m < n}$$

This rule is valid, but it's not as good as we could do. To see why, consider another example,

$F =$ `Function f -> f ({x=5; y=6; z=3}) + f({x=6; y=4})`.

Here the function `f` should, informally, take a record with at least `x` and `y` fields, but also should accept records where additional fields are present. Let us try to type the function F.

F : `({x:Int; y:Int} -> Int) -> Int`

Consider the application $F\ G$ for

$G =$ `Function r -> r.x + r.x`.

If we were to typecheck G, we would end up with G : {`x:Int`} `->` `Int`, which does not exactly match F's argument, {`x:Int; y:Int`} `->` `Int`, and so typechecking $F\ G$ will fail even though it does not cause a runtime error.

In fact we *could* have given G a type {`x:Int; y:Int`} `->` `Int`, but its too late to know that was the type we should have used back when we typed G . The *Sub-Rec*$_0$ rule is of no help here either. What we need is a rule that says that a function with a record type argument may have fields *added* to its record argument type, as those fields will be ignored:

$$\textit{(Sub-Function}_0\textit{)} \qquad \frac{\Gamma \vdash e : \{l_1 : \tau_1;\ \ldots;\ l_n : \tau_n\}\ \texttt{->}\ \tau}{\Gamma \vdash e : \{l_1 : \tau_1;\ \ldots;\ l_n : \tau_n;\ \ldots;\ l_m : \tau_m\}\ \texttt{->}\ \tau}$$

Using this rule, $F\ G$ will indeed typecheck. The problem is that we still need other rules. Consider records inside of records:

```
{pt = {x=4; y=5}; clr = 0} : {pt:{x:Int}; clr:Int}
```

should still be a valid typing since the `y` field will be ignored. However, there is no type rule allowing this typing either.

6.5.2 The STF♭R Type System: TF♭ with Records and Subtyping

By now it should be clear that the strategy we were trying to use above can never work. We would need a different type rule for every possible combination of records and functions!

The solution is to have a separate set of **subtyping** rules just to determine when one type can be used in the place of another. $\tau <: \tau'$ is read "τ is a subtype of τ'," and means that an object of type τ may also be considered an object of type τ'. The rule added to the **TF♭** type system (along with the record rules of **TF♭SRX**) is

$$\textit{(Sub)} \qquad \frac{\Gamma \vdash e : \tau, \quad \vdash \tau <: \tau'}{\Gamma \vdash e : \tau'}$$

We also need to make our subtyping operator reflexive and transitive. This can be accomplished with the following two rules.

$$\textit{(Sub-Refl)} \qquad \frac{}{\vdash \tau <: \tau}$$

$$\textit{(Sub-Trans)} \qquad \frac{\vdash \tau <: \tau', \quad \vdash \tau' <: \tau''}{\vdash \tau <: \tau''}$$

Our rule for subtyping records needs to do two things. It needs to ensure that if a record B is the same as record A with some additional fields, then B is a subtype of A. It also needs to handle the case of records within records. If B's fields are all subtypes of A's fields, then B should also be a subtype of A. We can reflect this concisely in a single rule as follows.

(Sub-Record)

$$\frac{\vdash \tau_1 <: \tau_1', \ldots, \tau_n <: \tau_n'}{\vdash \{l_1 : \tau_1;\ \ldots;\ l_n : \tau_n;\ \ldots;\ l_m : \tau_m\} <: \{l_1 : \tau_1';\ \ldots;\ l_n : \tau_n'\}}$$

The function rule must also do two things. If functions A and B are equivalent except that B returns a subtype of what A returns, then B is a subtype of A. However, if A

and B are the same except that B's argument is a subtype of A's argument, then A *is a subtype of* B. Simply put, for a function to be a subtype of another function, it has to take less and give more. The rule should make this clear:

$$\textit{(Sub-Function)} \qquad \frac{\tau_0' <: \tau_0, \quad \tau_1 <: \tau_1'}{\tau_0 \texttt{ -> } \tau_1 <: \tau_0' \texttt{ -> } \tau_1'}$$

From our discussions and examples in the previous section, it should be clear that this more general set of rules will work.

6.5.3 Implementing an STF♭R Type Checker

Automated typechecking of **STF♭R** is actually quite difficult. There are two ways to make the task easier. The first is to add more explicit type decoration to help the typechecker. The second it to completely infer the types in a *constraint* form, a topic covered in Section 6.7.

Here, we briefly sketch how the `typecheck` function for **STF♭R** may be written. The **TF♭** typechecker requires certain types to be identical, for example, the function domain type must be identical to the type of the function argument in an application $e\ e'$.

$$\textit{(Application)} \qquad \frac{\Gamma \vdash e : \tau \texttt{ -> } \tau', \quad \Gamma \vdash e' : \tau}{\Gamma \vdash e\ e' : \tau'}$$

In **STF♭R** at this point, we need to see if subtyping is possible. `typecheck(`e'`)` returns τ'' and then $\tau'' <: \tau$ is checked via a function `areSubtypes(`τ'', τ`)`. This produces a valid proof by the *Sub* rule. Other rules where the **TF♭** rules require a type match similarly are generalized to allow the *Sub* rule to be used.

Exercise 6.6. Implement the `areSubtypes` function.

6.5.4 Subtyping in Other Languages

It is interesting to see how subtyping is used in other languages. Consider, for example, Java and C++. In these languages, *subclassing* is the main form of subtyping. A subclass is a subtype of the class from which it extends. In Java, a class is also a subtype of any interfaces it implements.

Java and C++ are thus more restrictive. Suppose there were classes with structure `{x:Int; y:Int; color:Int}` and `{x:Int; y:Int}` that weren't one of the two cases above (that is, the prior is not a subclass of the latter, and the two share no common interface). The two classes, then, are not subtypes in Java or C++, but they are in **STF♭R**. Declared subtyping has the advantage, however, that subtype relationships do not have to be completely inferred.

OCaml objects are more flexible in that there is no restriction to a hierarchy, but it's also less flexible in that there is really no object polymorphism— an explicit coercion of a `ColorPoint` to a `Point` is required. There are several research languages with type inference and subtyping, but the types are often complex or hard to read [9].

6.6 Type Inference and Polymorphism

Type inference was originally discovered by Robin Milner, the original creator of ML, and independently by the logician J. Roger Hindley. The key idea of Milner's "Algorithm W" is to initially give all variables arbitrary types, `'a`, and then to *unify* or equate the types, if indicated by the program. For example, if the application `f x` has type `'a -> 'b` and `x` has type `'c`, we may equate `'a` and `'c`.

We will look at full type inference, that is, type inference on programs with no explicit type information.

6.6.1 Type Inference and Polymorphism

Type inference goes hand-in-hand with **parametric polymorphism** (often called "**generic types**"). Consider the function `Function x -> x`. Without polymorphism, what type can be inferred for this function? `Int -> Int` is a flawed answer, because the function could be used in a context where it is passed a boolean. With type inference we need to achieve something called a principal type.

Definition 6.3. *(Principal Type) A* ***principal type*** τ *for expression* e *(where* $\vdash e : \tau$*) has the following property. For any other type* τ' *such that* $\vdash e : \tau'$*, for any context* C *for which* $\vdash C[e : \tau'] : \tau''$ *for any* τ''*, then* $\vdash C[e : \tau] : \tau'''$ *as well, for some* τ'''*.*

What principality means is no other typing will let more uses of the program type-check, so the principal typing will always be best. The desired property of our type inference algorithm is that it will always infer principal types. An important corollary is that with a principal-type algorithm, we know inference will never "get in the way" of the programmer by, for example, inferring `Bool -> Bool` for the identity function when the programmer wants to use it on `Int -> Int`.

OCaml infers the type `'a -> 'a` for the identity function, which can be shown to be a principal type for it. In fact, OCaml type inference always infers principal types.

6.6.2 An Equational Type System: EF♭

We are going to present type inference in a nonstandard way. Milner's "Algorithm W" *eagerly* unifies `'a` and `'c`: it replaces one with the other everywhere. We will present **equational inference**, in which we lazily accumulate equations like $\texttt{'a} = \texttt{'c}$, and then solve the system of equations at the end of the algorithm.

We will study **EF♭**, a simple, equationally typed version of **F♭**. **EF♭** uses the same grammar for expressions as the **F♭** language did, since **EF♭** does not have any type decorations. The **EF♭** types are

$$\begin{array}{lcll} \tau & ::= & \texttt{Int} \mid \texttt{Bool} \mid \tau \texttt{ -> } \tau \mid \alpha & \textit{types} \\ \alpha & ::= & \texttt{'a} \mid \texttt{'b} \mid \ldots & \textit{type variables} \end{array}$$

EF♭ types during inference are going to include an extra set of *constraining equations*, E, which constrain the behavior of the type variables. Type judgments for **EF♭** are thus of the form $\Gamma \vdash e : \tau \backslash E$, the same as before but tacking a set of equations on the side. Each member of E is an equation like $\texttt{'a} = \texttt{'c}$. Equational types will be used to aid inference. Here is an outline of the overall approach.

1. Infer equational types for the whole programs.
2. If the equations are inconsistent, pronounce that there is a type error.
3. If the equations are consistent, simplify them to give an inferred type.

It is a fact that if there are no inconsistencies in the equations, they can always be simplified to give an equation-free type.

Definition 6.4. *(Equational Type) An* ***equational type*** *is a type of the form*

$$\tau \backslash \{\tau_1 = \tau_1', \ldots, \tau_n = \tau_n'\}$$

Each $\tau = \tau'$ is an equation on types, meaning τ and τ' have the same meaning as types. We will let E mean some arbitrary set of type equations. For instance,

$$\texttt{Int -> 'a} \backslash \{\texttt{'a} = \texttt{Int -> 'a1}, \texttt{'a1} = \texttt{Bool}\}$$

is an equational type. If you think about it, this is really the same as the type

`Int -> Int -> Bool`.

This is known as *equation simplification* and is a step we will perform in our type inference algorithm. It is also possible to write meaningless types such as

$$\texttt{Int -> 'a} \backslash \{\texttt{'a} = \texttt{Int -> 'b}, \texttt{'a} = \texttt{Bool}\}$$

which cannot be a type since it implies that functions and booleans are the same type. Such equation sets are deemed *inconsistent*, and will be equated with failure of the type inference process. There are also possibilities for circular (self-referential) types that don't quite look inconsistent:

$$\texttt{Int -> 'a} \backslash \{\texttt{'a} = \texttt{Int -> 'a }\}$$

OCaml disallows such types, and we will also disallow them initially. These types can't be simplified away, and that is the main reason why OCaml disallows them: users of the language would have to see some type equations.

The EF♭ Type Rules

The **EF**♭ system is the following set of rules. Note that Γ serves the same role as it did in the **TF**♭ rules. It is a type environment that binds variables to simple (non-equational) types. Our axiomatic rules look much like the **TF**♭ rules.

(Hypothesis) $$\frac{}{\Gamma \vdash x : \tau \backslash \emptyset \text{ for } \Gamma(x) = \tau}$$

(`Int`) $$\frac{}{\Gamma \vdash n : \texttt{Int} \backslash \emptyset \text{ for } n \text{ an integer}}$$

(`Bool`) $$\frac{}{\Gamma \vdash b : \texttt{Bool} \backslash \emptyset \text{ for } b \text{ a boolean}}$$

The rules for `+`, `-`, and `=` also look similar to their **TF**♭ counterparts, but now we must take the union of the equations of each of the operands to be the set of equations for the type of the whole expression, and add an equation to reflect the type of the operands.

(`+`) $$\frac{\Gamma \vdash e : \tau \backslash E, \quad \Gamma \vdash e' : \tau' \backslash E'}{\Gamma \vdash e \texttt{ + } e' : \texttt{Int} \backslash E \cup E' \cup \{\tau = \texttt{Int}, \tau' = \texttt{Int}\}}$$

(`-`) $$\frac{\Gamma \vdash e : \tau \backslash E, \quad \Gamma \vdash e' : \tau' \backslash E'}{\Gamma \vdash e \texttt{ - } e' : \texttt{Int} \backslash E \cup E' \cup \{\tau = \texttt{Int}, \tau' = \texttt{Int}\}}$$

(`=`) $$\frac{\Gamma \vdash e : \tau \backslash E, \quad \Gamma \vdash e' : \tau' \backslash E'}{\Gamma \vdash e \texttt{ = } e' : \texttt{Bool} \backslash E \cup E' \cup \{\tau = \texttt{Int}, \tau' = \texttt{Int}\}}$$

The *`And`*, *`Or`*, and *`Not`* rules are defined in a similar way. The rule for `If` is also similar to the **TF**♭ *`If`* rule. Notice, though, that we do not immediately infer a certain type like we did in the previous rules. Instead, we infer a type α, and equate α to the types of the `Then` and `Else` clauses.

(`If`)
$$\frac{\Gamma \vdash e : \tau \backslash E, \quad \Gamma \vdash e' : \tau' \backslash E', \quad \Gamma \vdash e'' : \tau'' \backslash E''}{\Gamma \vdash (\texttt{If } e \texttt{ Then } e' \texttt{ Else } e'') : \alpha \backslash E \cup E' \cup E'' \cup \{\tau = \texttt{Bool}, \tau' = \tau'' = \alpha\}}$$

Finally, we are ready for the function and application rules. Functions no longer have explicit type information, but we may simply choose a fresh type variable `'a` as the function argument, and include it in the equations later. The application rule also picks a fresh type variable `'a` type, and adds an equation with `'a` as the right hand side of a function type. The rules should make this clear.

(Function) $$\frac{\Gamma, x : \alpha \vdash e : \tau \backslash E}{\Gamma \vdash (\texttt{Function } x \texttt{ -> } e) : \alpha \texttt{ -> } \tau \backslash E}$$

(Application) $$\frac{\Gamma \vdash e : \tau \backslash E, \quad \Gamma \vdash e' : \tau' \backslash E'}{\Gamma \vdash e\ e' : \alpha \backslash E \cup E' \cup \{\tau = \tau' \texttt{ -> } \alpha\}}$$

These rules almost directly define the equational type inference procedure: the proof can pretty much be built from the bottom (leaves) on up. Each equation added denotes two types that should be equal.

Solving the Equations

One thing that should be immediately clear from the **EF**♭ type rules is that *any* syntactically correct program may be typed by these rules. Whether or not a program is *well-typed* is not determined until we actually solve the system of equations in the equational type of the entire program. The **EF**♭ type rules are really just accumulating constraints.

Solving the equations involves two steps. First we compute the *closure* of the equations, producing new equations that hold by transitivity, etc. Next, we check for any inconsistent equations, such as `Int` = `Bool` that denote type errors.

The following algorithm computes the equational closure of set E.

- For each equation of the form τ_0 `->` $\tau_0' = \tau_1$ `->` τ_1' in E, add $\tau_0 = \tau_1$ and $\tau_0' = \tau_1'$ to E.
- For each set of equations $\tau_0 = \tau_1$ and $\tau_1 = \tau_2$ in E, add the equation $\tau_0 = \tau_2$ to E (by transitivity).
- Repeat (1) and (2) until no more equations can be added to E.

Note that we will implicitly use the symmetric property on these equations, and so there is no need to add $\tau_1 = \tau_0$ for every equation $\tau_0 = \tau_1$.

The closure serves to uncover inconsistencies. For instance,

Closure({`'a` = `Int -> 'b`, `'a` = `Int -> Bool`, `'b` = `Int`}) =
{`'a` = `Int -> 'b`, `'a` = `Int -> Bool`,
`'b` = `Int`, `Int -> 'b` = `Int -> Bool`,
`Int` = `Int`, `'b` = `Bool`, `Int` = `Bool`},

directly uncovering the inconsistency `Int` = `Bool`.

The closure of E can be computed in polynomial time. After computing the closure, the constraints are consistent if

1. No immediate inconsistencies are uncovered, such as `Int` = `Bool`, `Bool` = τ `->` τ', or `Int` = τ `->` τ'.
2. No self-referential equations exits (we will deal with this issue shortly).

If the equations are consistent, the next step is to solve the equational constraints. We do this by substituting type variables with actual types. The algorithm is as follows. Given $\tau \backslash E$,

1. Replace some type variable α in τ with τ', provided $\alpha = \tau'$ or $\tau' = \alpha$ occurs in E and either
   - τ' is not a type variable, or
   - τ' is a type variable α' which lexographically succeeds α.
2. Repeat (1) until no more such replacements are possible.

Notice that step (1) considers the symmetric equivalent of each equation, which is why we didn't include them in the closure. The algorithm has a flaw though: the replacements may continue forever. This happens when E contains a circular type. Recall the example of a self-referential type

`Int -> 'a`$\backslash\{$`'a = Int -> 'a`$\}$.

Trying to solve these constraints results in the nonterminating chain

`Int -> Int -> 'a`$\backslash\{$`'a = Int -> 'a`$\}$
`Int -> Int -> Int -> 'a`$\backslash\{$`'a = Int -> 'a`$\}$
`Int -> Int -> Int -> Int -> 'a`$\backslash\{$`'a = Int -> 'a`$\}$
...

The solution is to check for such cycles before trying to solve the equations. The best way to do this it to phrase the problem in graph-theoretical context. Specifically, we define a directed graph G in which the nodes are the the type variables in E. There is a directed edge from `'a` to `'b` if `'a` $= \tau$ is an equation in E and `'b` occurs in τ.

We raise a `typeError` if there is a cycle in G for which there is at least one edge representing a constraint that isn't just between type variables (`'a = 'b`).

In summary, our entire **EF**$\flat$ type inference algorithm is as follows. For expression e,

1. Produce a proof of $\vdash e : \tau\backslash E$ by applying the **EF**$\flat$ type rules. Such a proof always exists.
2. Extend E by computing its closure.
3. Check if E is immediately inconsistent. If so, raise a `typeError`.
4. Check for cycles in E using the algorithm described above. If there is a cycle, raise a `typeError`.
5. Solve E by the above equation solution algorithm. This algorithm will always terminate if there are no cycles in E.
6. Output: the solution type τ' for e produced by the solution algorithm.

Theorem 6.2. *The typings produced by the above algorithm are always principal.*

The proof of this theorem is beyond the scope of this book.

Let's conclude with an example of the type inference algorithm in action. Suppose we want to infer the type of the expression

```
(Function x -> If x Then 3 Else 4) False
```

First we produce the following proof.

By the *Application* rule,
$\vdash$ (`(Function x -> If x Then 3 Else 4) False`) :
`'c`$\backslash\{$`'a` $=$ `Bool, Int` $=$ `'b, 'a -> 'b` $=$ `Bool -> 'c`$\}$
Because, by the ***`Function`*** rule,
`(Function x -> If x Then 3 Else 4)` :
`'a -> 'b`$\backslash\{$`'a` $=$ `Bool, Int` $=$ `'b`$\}$
Because, by the ***`If`*** rule,
`x : 'a` $\vdash$ `If x Then 3 Else 4 : 'b`$\backslash\{$`'a` $=$ `Bool, Int` $=$ `'b`$\}$
Because by the ***`Int`*** and *Hypothesis* rules,
`x : 'a` $\vdash$ `x : 'a`$\backslash\emptyset$,
`x : 'a` $\vdash$ `3 : Int`$\backslash\emptyset$, and
`x : 'a` $\vdash$ `4 : Int`$\backslash\emptyset$
And, by the ***`Bool`*** rule,
$\vdash$ `False : Bool`$\backslash\emptyset$

Given the proof of

$\vdash$ (`(Function x -> If x Then 3 Else 4) False`) :
`'c`$\backslash\{$`'a` $=$ `Bool, Int` $=$ `'b, 'a -> 'b` $=$ `Bool -> 'c`$\}$

we compute the closure of the set of equations to be

$\{$`'a` $=$ `Bool, Int` $=$ `'b, 'a -> 'b` $=$ `Bool -> 'c, 'b` $=$ `'c, Int` $=$ `'c`$\}$

The set is not immediately inconsistent, and does not contain any cycles. Therefore, we solve the equations. In this case, τ contains only `'c`, and we can replace `'c` with `Int`. We output `Int` as the type of the expression, which is clearly correct.

6.6.3 PEF♭: EF♭ with `Let` Polymorphism

After all our work on **EF**♭, we still don't have polymorphism, we only have type variables. To illustrate this, consider the function

```
Let x = Function y -> y In (x True); (x 0)
```

Recalling our encoding of `Let` as a function, this expression is equivalent to

```
(Function x -> (x True); (x 0)) (Function y -> y)
```

In OCaml, such programs typecheck fine. Different uses of `Function y -> y` can have different types. Consider what **EF**$\flat$ would do when typing this expression, though.

$\vdash$ (`Function x -> (x True); (x 0)`) :
`'a -> 'c`$\backslash\{$`'a` $=$ `Bool -> 'b`, `'a` $=$ `Int -> 'c`, $\ldots\}$

But when we compute the closure of this equational type, we get the equation `Int` = `Bool`! What went wrong? The problem in this case is that each use of `x` in the body used the same type variable `'a`. In fact, when we type `Function y -> y`, we know that `'a` can be anything, so for different uses, `'a` can be different things. We need to build this intuition into our type system to correctly handle cases like this. We define such a type system in **PEF**$\flat$, which is **EF**$\flat$ with `Let` and `Let`-polymorphism.

PEF$\flat$ has a special `Let` typing rule, in which we allow a new kind of type in Γ: $\forall\alpha_1 \ldots \alpha_n.\ \tau$. This is called a **type schema**, and may only appear in Γ. An example of a type schema is $\forall\alpha.\ \alpha$ `->` α. Note that the type variables $\alpha_1 \ldots \alpha_n$ are considered to be *bound* by this type expression.

The new rule for `Let` is

$$\textit{(Let)} \qquad \frac{\Gamma \vdash e : \tau \backslash E, \quad \Gamma, x : \forall\alpha_1 \ldots \alpha_n.\ \tau' \vdash e' : \tau'' \backslash E'}{\Gamma \vdash (\texttt{Let}\ x\ \texttt{=}\ e\ \texttt{In}\ e') : \tau'' \backslash E'}$$

where τ' is a solution of $\vdash e : \tau \backslash E$ using the above algorithm, and τ' has free type variables $\alpha_1 \ldots \alpha_n$ that do not occur in Γ.

Notice that since we are invoking the simplification algorithm in this rule, it means the full algorithm is not the clean 3-pass infer-closure-simplify form give above: the rules need to call close-simplify on some sub-derivations.

We also need to add an axiom to ensure that a fresh type variable is given to each `Let` usage. The rule is

$$\textit{(Let Inst.)} \qquad \frac{}{\Gamma, x : \forall\alpha_1 \ldots \alpha_n.\ \tau' \vdash x : R(\tau') \backslash \emptyset}$$

where $R(\tau')$ is a renaming of the variables $\alpha_1 \ldots \alpha_n$ to fresh names. Since these names are fresh each time `x` is used, the different uses won't conflict like above.

It will help to see an example of this type system in action. Let's type the example program from above:

```
Let x = Function y -> y In (x True); (x 0)
```

We have

$\vdash$ `Function y -> y : 'a -> 'a`$\backslash\emptyset$

This constraint set trivially has the solution type `'a -> 'a`. Thus, we then typecheck the `Let` body under the assumption that `x` has type $\forall$`'a.'a -> 'a`.

`x :` $\forall$`'a.'a -> 'a` $\vdash$ `x : 'b -> 'b`$\backslash\emptyset$

by the *Let-Inst* rule. Then

`x :` $\forall$`'a.'a -> 'a` $\vdash$ `x True : 'c`$\backslash\{$`'b -> 'b = Bool -> 'c`$\}$

Similarly,

`x :` $\forall$`'a.'a -> 'a` $\vdash$ `x 0 : 'e`$\backslash\{$`'d -> 'd = Int -> 'e`$\}$

The important point here is that this use of `x` gets a different type variable, `'d`, by the *Let-Inst* rule. Putting the two together, the type is something like

`x :` $\forall$`'a.'a -> 'a` $\vdash$ `x True; x 0 :`
`'e`$\backslash\{$`'b -> 'b = Bool -> 'c, 'd -> 'd = Int -> 'e`$\}$

which by the ***`Let`*** rule then produces

$\vdash$ `(Let x = Function y -> y In (Function x -> x True; x 0)) :`
`'e`$\backslash\{$`'b -> 'b = Bool -> 'c, 'd -> 'd = Int -> 'e`$\}$

Since `'b` and `'d` are different variables, we don't get the conflict we got previously.

6.7 Constrained Type Inference

There was a reason why we presented Hindley-Milner type inference in the form above: if we replace equality constraints by subtyping constraints, <:, we can perform constrained type inference. To understand why it is useful to perform this generalization, it is easiest to just look at the rules.

F♭ is not the best system to show off the power of replacing equality with subtyping. Since the language does not have records, there is not any interesting subtyping that could happen. To show the usefulness of subtyping, we thus define the constraints in an environment where we have records, **F**♭**R**. **F**♭**R** plus constraints is **CF**♭**R**. We can contrast **CF**♭**R** with the **EF**♭**R** language which we did not study but is simply **EF**♭ with support for records. Instead of types $\tau \backslash E$ for a set of equations E, **CF**♭**R** has types

$$\tau \backslash \{\tau_1 <: \tau_1', \ldots, \tau_n <: \tau_n'\}$$

CF♭**R** has the following set of type rules. These are direct generalizations of the **EF**♭ rules, replacing = by <:. The <: is always in the direction of information flow. We let C represent a set of subtyping constraints.

(Hypothesis)
$$\frac{}{\Gamma \vdash x : \tau \backslash \emptyset \text{ for } \Gamma(x) = \tau}$$

(`Int`)
$$\frac{}{\Gamma \vdash n : \texttt{Int} \backslash \emptyset \text{ for } n \text{ an integer}}$$

(`Bool`)
$$\frac{}{\Gamma \vdash b : \texttt{Bool} \backslash \emptyset \text{ for } b \text{ a boolean}}$$

(`+`)
$$\frac{\Gamma \vdash e : \tau \backslash C, \quad \Gamma \vdash e' : \tau' \backslash C'}{\Gamma \vdash e \texttt{ + } e' : \texttt{Int} \backslash C \cup C' \cup \{\tau <: \texttt{Int}, \tau' <: \texttt{Int}\}}$$

(`-`)
$$\frac{\Gamma \vdash e : \tau \backslash C, \quad \Gamma \vdash e' : \tau' \backslash C'}{\Gamma \vdash e \texttt{ - } e' : \texttt{Int} \backslash C \cup C' \cup \{\tau <: \texttt{Int}, \tau' <: \texttt{Int}\}}$$

(`=`)
$$\frac{\Gamma \vdash e : \tau \backslash C, \quad \Gamma \vdash e' : \tau' \backslash C'}{\Gamma \vdash e \texttt{ = } e' : \texttt{Bool} \backslash C \cup C' \cup \{\tau <: \texttt{Int}, \tau' <: \texttt{Int}\}}$$

(`If`)
$$\frac{\Gamma \vdash e : \tau \backslash C, \quad \Gamma \vdash e' : \tau' \backslash C', \quad \Gamma \vdash e'' : \tau'' \backslash C'',}{\Gamma \vdash (\texttt{If } e \texttt{ Then } e' \texttt{ Else } e'') : \alpha \backslash C \cup C' \cup C'' \cup \{\tau <: \texttt{Bool}, \tau' <: \alpha, \tau'' <: \alpha\}}$$

(Function)
$$\frac{\Gamma, x : \alpha \vdash e : \tau \backslash C}{\Gamma \vdash (\texttt{Function } x \texttt{ -> } e) : \alpha \texttt{ -> } \tau \backslash C}$$

(Application)
$$\frac{\Gamma \vdash e : \tau \backslash C, \quad \Gamma \vdash e' : \tau' \backslash C'}{\Gamma \vdash e\ e' : \alpha \backslash C \cup C' \cup \{\tau <: \tau' \texttt{ -> } \alpha\}}$$

The two rules we have not seen in **EF**♭ are the *Record* and *Projection* rules. There is nothing particularly special about these rules, however.

$$\textit{(Record)} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \backslash C_1, \ldots, \Gamma \vdash e_n : \tau_n \backslash C_n}{\Gamma \vdash \{l_1\texttt{=}e_1\texttt{;}\ \ \ldots\texttt{;}\ \ l_n\texttt{=}e_n\} : \{l_1 : \tau_1\texttt{;}\ \ \ldots\texttt{;}\ \ l_n : \tau_n\} \backslash C_1 \cup \ldots \cup C_n}$$

$$\textit{(Projection)} \qquad \frac{\Gamma \vdash e : \tau \backslash C}{\Gamma \vdash e\texttt{.}l : \alpha \backslash \{\tau <: \{l : \alpha\}\} \cup C}$$

As with **EF**♭, these rules almost directly define the type inference procedure and the proof can pretty much be built from the bottom up.

The complete type inference algorithm is as follows. Given an expression e,

1. Produce a proof of $\vdash e : \tau \backslash C$ using the above type rules. Such a proof always exists.
2. Extend C by computing the closure as described below.
3. If C is immediately inconsistent, raise a `typeError`.
4. Check C for cycles as described below. If C contains a cycle, raise a `typeError`.
5. The inferred type is $e : \tau \backslash C$.

The algorithms for computing the closure of C and doing cycle detection are fairly obvious generalizations of the **EF**♭ algorithms. Closure is computed as follows.

1. For each constraint $\{l_1 : \tau_1, \ldots, l_n : \tau_n, \ldots, l_m : \tau_m\} <: \{l_1 : \tau'_1, \ldots, l_n : \tau'_n\}$ in C, add $\tau_1 <: \tau'_1$, $\ldots$, $\tau_n <: \tau'_n$ to C.
2. For each constraint τ_0 `->` $\tau'_0 <: \tau_1$ `->` τ'_1 in C, add $\tau_1 <: \tau_0$ and $\tau'_0 <: \tau'_1$ to C.
3. For constraints $\tau_0 <: \tau_1$ and $\tau_1 <: \tau_2$, add $\tau_0 <: \tau_2$ to C (by transitivity).
4. Repeat until no more constraints can be added.

A constraint set is *immediately inconsistent* if $\tau <: \tau'$ and τ and τ' are different kinds of type (function and record, `Int` and function, etc), or two records are ordered by $<:$ and the right record has a field the left record does not.

To perform cycle detection in C, we use the following algorithm. Define a directed graph G where nodes are type variables in C. There is an edge from a `'a` node to a `'b` node if there is an equation `'a` $<: \tau'$ in C, and `'b` occurs in τ'. Additionally, there is an edge from `'b` to `'a` if $\tau' <:$ `'a` occurs in C and `'b` occurs in τ'. C has a cycle if and only if G has a cycle.

It seems that there is a major omission in our constrained type inference algorithm: we never solve the constraints! The algorithm is correct, however. The reason we don't want to solve the constraints is that any substitution proceeds with possible loss of generality.

Consider, for example, a constraint `'a` $<: \tau$, and the possibility of substituting `'a` with τ. This precludes the possibility that the `'a` position be a subtype of τ, as the substitution in effect asserts the equality of `'a` and τ. In simpler terms, we need to keep the constraints around as part of the type. This is the main weakness of constrained type systems; the types include the constraints, and are therefore difficult to read and understand.

We have the same shortcomings as in the equational case at this point: there is as of yet no polymorphism. The solution used in the equational case won't work here, as it required the constraints to be solved.

The solution is to create constrained polymorphic types

$$\forall \alpha_1, \ldots, \alpha_n.\ \tau \backslash C$$

in the assumptions Γ, in place of the polymorphic types (type schema) we had in the equational version. The details of this process are quite involved, and we will not go into them. Constrained polymorphic types make very good object types, since polymorphism is needed to type inheritance.

Chapter 7

Concurrency

A concurrent computation is working on more than one thing at once. We assume some familiarity with concurrency but provide a brief overview to get us going. The Wikipedia article on parallel computing provides a more detailed overview.

7.1 Overview

Concurrent execution can be loosely grouped into three implementation categories, in order of loosest to tightest coupling. *Distributed computation* is computation on multiple computers which share no memory and are sending messages between each other to communicate data. There is still a wide range of how tightly these computers can be decoupled. *Grid computing* is distributed computing where the Internet is the communication medium. *Cluster computing* is over a fast LAN network. Massive Parallel Processing (MPP) involves specialized communication hardware for very high communication bandwidth. *Distributed shared memory* is the case where different processes are running on different processors but sharing some special memory via a memory bus. Lastly, *multithreaded computation* is the case where multiple threads of execution share a single memory which is local. Multithreaded computations may run on a single (core) CPU, meaning the concurrent execution is an illusion achieved by interleaving the execution steps of two threads, or if the computer has multiple cores there can be true concurrent execution of a multithreaded program. The Threads Wikipedia article clarifies how threads and processes differ.

All of the above models still support independent foci of control. That is the primary focus of our study in this chapter – it captures a wide range of models and they are the most elegant forms of concurrent architecture. There are several other models that we are not addressing. *Vector processors* are computers that can do an operation on a whole array in one step. These architectures used to be called SIMD (Single Instruction Multiple Data). *Stream processors* are the modern version of vector processors which can work on more than just arrays in parallel, for example sparse arrays. The *GPGPU* (General Purpose Graphics Processing Units) is a recent version of a stream processor which arose as a generalization of specialized graphics processors. Lastly, *FPGA's* are Field Programmable Circuits: you can create your own (parallel) logic circuitry on the fly.

Historically, concurrent programming arose as library extensions to existing languages. Early models included UNIX sockets for IPC, and process forking in C. While some concurrency is easily programmable via library extensions to an underlying sequential language, some aspects of concurrency are fundamental enough that it is important to integreate concurrency into the programming language.

The standard model these days for Java, C, etc concurrency is via multithreading. Multithreaded programming is coming on in a big way – newer CPUs have multiple cores and can run multiple threads simultaneously. Mulithreaded programming unfortunately is also a disaster waiting to happen – the programs are just too hard to debug. The root of the problem is there are too many possible cases of interleaving of the different threads that can occur when accessing the shared memory. Some of them may show up in testing, but many of them will only show up after deployment.

The primary "Bad Things" that can occur in multithreaded programming include the following. A **race condition** is when two operations are interleaved and leave the data in an inconsistent state. For example, a simultaneous variable update where one thread set the high word of a double and another thread set the low word due to near- simultaneous access – the double's value is not a sensible value from either thread. **Deadlock** is when threads may need to wait for resources to free up (they were locked to begin with to prevent race conditions), and can get in a **cycle** of waiting: A waits for B waits for C waits for A.

Race conditions may be avoided via the use of locks of various kinds. **Monitors** are regions of mutual exclusion in the source program, only one thread can execute a monitor block at any one point. They are similar to the `synchronized` keyword of Java. A **Semaphore** is a low-level locking mechanism: grab a lock before a critical operation; block if someone else has lock; once you have the lock you know you are the only one accessing; free it when you are done. General semaphores allow n threads to simultaneously access a critical region, not just one.

Atomicity is another key design concept of concurrent programming languages. An **atomic region** is a region of code that may have been interleaved with other thread executions, but you can't tell – it *always appears to have run atomically* , in one step. The more atomicity you can get in your language design the fewer interleavings need to be considered in debugging and the fewer bugs. The Sun Java Concurrency Tutorial provides more details on the model.

7.1.1 The Java Concurrency Model

Let us briefly review the Java approach to concurrency. Java implementats the standard notion of thread: memory is shared between concurrent threads of control (i.e., each thread has its own runtime stack but only one heap). The `synchronized` keyword is used to declare zones of mutual exclusion; they are a form of monitor. There are several variants: `synchronized` methods or blocks of code can both be defined. When a synchronized method/block is running, no other synchronized method/block for that object can start from another thread - any such threads would have to wait until the currently running method/block finishes. An advantage of Java's `synchronized` is how it is *object-based*: locks are per-object and this is arguably a good level of abstraction for locking. The disadvanatge of Java however is monitors give only mutual exclusion,

not atomicity.

The Java concurrency model has some other nice features which we briefly review here. Much of this is found in the `java.util.concurrent` package.

- Atomic integers, floats, etc - there will never be any race conditions on setting or getting the value from `AtomicInteger` etc.
- Locks - `java.util.concurrent.locks.Lock`
- Concurrent collections - e.g. `ConcurrentHashMap` which supports concurrent add/lookup atomically.

7.2 The Actor Model and AF♭V

We will study one model of concurrent programming in more detail here, the **Actor Model**. Actors are a simple, elegant model of concurrent programming. It is to some degree the "functional programming" analogy in the concurrent programming world, and Actors in fact fit very well with functional programming style. The Erlang programming language [1] is a real-world language with exactly that structure: a functional basis with an actor concurrency layar on top.

The actor model was originally developed by Karl Hewitt at MIT in the 1970's. The Actor model Wikipedia entry has a good historical overview and description, and we now briefly summarize the key points. Each actor is an autonomous, distributed agent, and has a name that is not forgeable. All messaging is *asynchronous* – sending actors never wait for a reply from the receiving actor. The arrival order of the messages is nondeterministic, they may arrive in a different order they were sent in. Actors can be thought of as a non-shared-memory model with respect to our earlier classification: all communication between actors is via explicit messaging, not by implicit communication through a shared variable in a common memory. Along with their local computation actions and sending of messages, actors can create other actors. If an actor is busy when a message arrives is it put in a message queue – only one message is processed at a time by an actor. Concurrency is achieved by the fact that many actors can be processing their messages in parallel, not by concurrent behavior of an individual actor. In our idealized model we assume there are no faults: all messages eventually arrive (but, they may take arbitrarily long to do so). Each actor is a *reactive system*: normally the actor is idle, is wakes up and process a single message in a finite amount of time, and goes back to sleep until another message comes in. This is the life of the actor, always and forever reacting to events from the outside.

One of the biggest advantages of the actor model is the built-in *Atomicity*: multiple actors can be running in parallel, but any interleaved run is equivalent to a run where each actor runs all of its steps in one big step. That is because actors can never accept messages in the middle of processing their original message.

In this section we define **AF♭V**, an actor layer on top of the **F♭V** language. We need the "**V**" because we will use variants to define messages. Recall **F♭V**variants are like OCaml's inferred variants - `'foo(4)` is the variant `foo` with argument `4`. Notice how we can also view this as the **message** `foo` with argument `4`. **F♭V**variants always have exactly one argument only, for simplicity.

7.2.1 Syntax of AF♭V

The **AF♭V** expressions are the following.

$$\begin{array}{rcll} v & ::= & \ldots \text{ the } \mathbf{F}\flat\mathbf{V} \text{ values} \ldots \mid a & \textit{values} \\ e & ::= & \ldots \text{ the } \mathbf{F}\flat\mathbf{V} \text{ expressions} \ldots \mid e \leftarrow e \mid \texttt{Create}(e,e) \mid a & \textit{expressions} \end{array}$$

where a are taken from an unbounded set of actor names. They are like the cells c of **F♭S** in that they cannnot appear in source programs but can show up at runtime, and there are infinitely many unique ones; they are just names (nonces).

AF♭V syntax $e \leftarrow e'$ indicates a message send; it expects e to evaluate to an actor name, and sends that actor the message which is the value of e'. `Create`(e, e') creates an actor with behavior e, and with initial local data e'. e should evaluate to a function and that function is the (whole) code for the actor. The `Create` returns the (new) name of this new actor as its result.

Some features of **AF♭V** include the following. The code of an actor is nothing but one function – there is no state within an actor in the form of fields. At the end of processing each message, the actor goes idle; the behavior it is going to have upon receiving the next message is *the value at the end of the previous message send.* The latter point is the key to how actors can mutate over time: each message processing is purely functional, but at the end the actor gets to pick what state it wants to mutate to before processing the next message. This is an interesting method for mixing a bit of imperative programming into a pure functional model.

In **AF♭V** you must write your own explicit message dispatch code using the `Match` syntax of **F♭V**. We are here taking the dual approach to encoding to objects compared to the objects-as-records approach we used in **F♭OB**, here taking objects as functions and messages as variants. Lastly, in order to allow actors to know their own name, at creation time it is passed to them.

7.2.2 An Example

Before getting into the operational semantics lets do a simple example. Here is an actor that gets a start message and then counts down from its initial value to 0. Here, we use the term `Y` to represent the Y-combinator and the term `_` to represent a value of no importance (as with the empty record {} before).

```
Function myaddr ->
  Y (Function this -> Function localdata -> Function msg ->
      Match msg With
         ‘main(n) -> myaddr <- ‘count(n); this(_)
       | ‘count(n) -> If n = 0
           Then this(_)
           Else
              myaddr <- ‘count(n-1);
              this(_) /* set to respond to next message */
     )
```

Here is a code fragment that another actor could use to fire up a new actor with the above behavior and get it started. Suppose the above code we abbreviated `CountTenBeh`.

```
Let x = Create(CountTenBeh,_)  /* _ is the localdata - unused */
  In x <- ‘main(10)
```

Here is an alternative way to count down, where the `localdata` field holds the value, and its not in the message.

```
Function myaddr ->
  Y (Function this -> Function localdata -> Function msg ->
      Match msg With
        ‘count(_) ->
           If localdata = 0
             Then _
             Else
                myaddr <- ‘count(_);
                this(localdata - 1) /* set to respond to next msg */
     )
```

Suppose the above code was abbreviated `CountTenBeh2`; using it is then

```
Let x = create(CountTenBeh2,10)  /* 10 is  the localdata */
  In x <- ‘count(_)
```

The latter example is the correct way to give actors local data – in the former example the counter value had to be forwarded along every message.

Here is another usage fragment for the first example:

```
Let x = create(CountTenBeh,_)
  In x <- ‘main(10); x <- ‘main(5)
```

In this case the actor `x` will in parallel and independently counting down from 10 .. 0 and 5 .. 0 - these counts may also interleave in random ways. For the second example an analogue might be:

```
Let x = create(CountTenBeh2,10)
  In x <- ‘count(_); x <- ‘count(_)
```

This does nothing but get one more count message queued up; since the actor sends a new one out each time it gets one until 0, the effect will be to have a leftover message at the end.

7.2.3 Operational Semantics of Actors

The operational semantics for actors has two layers: the local computation within an actor, which is not too different than $\mathbf{F}\flat\mathbf{V}$, and the concurrent global stepping of all the actors. Lets start with the latter.

A *global state* G is a "soup" of the active actors and sent messages:

$$G ::= \begin{array}{l} \{\langle a, v\rangle \mid a \text{ is an actor name, } v \text{ is its behavior}\} \\ \cup \\ \{[a \leftarrow v] \mid a \text{ is an actor name, } v \text{ is the message sent to } a\} \end{array}$$

Actor systems can run forever, there is no notion of a final value. So, the system does *small steps* of computation:

$$G_1 \rightarrow G_2 \rightarrow G_3 \rightarrow \ldots$$

– this indicates *one step* of computation; in each single step *one* actor in the soup competely processed *one* message in the soup. We will define this $\rightarrow$ relation below. $\rightarrow^*$ is the reflexive, transitive closure of $\rightarrow$ – many steps of actor computation. In general this continues infinitely since actor systems may not terminate. The final meaning of a run of an actor system is this infinite stream of states.

7.2.4 The Local Rules

Lets start with the local rules. They are defined with a similar relation $\Rightarrow$ in $\mathbf{F\flat V}$ operational semantics, but the local executions additionally have *side effects* of the actors they create and messages they send. We will make any such side effects be *labels* on this arrow relation. So we make the local compuration relation be $\stackrel{S}{\Longrightarrow}$ – S here is the soup of effects. S is in fact the same syntactic form as a G: it contains two kinds of elements, $[a \leftarrow v]$ indicating a send to a of message v that this local actor performed over its execution, and $\langle a, v\rangle$ indicating a new actor it created, named a, with behavior (body) v.

Now let us look at the operational semantics rules for $\stackrel{S}{\Longrightarrow}$. Most of the rules are very minor changes to the corresponding $\mathbf{F\flat V}$ rule; we just give the `+` rule to show how the existing $\mathbf{F\flat V}$ rules must be tweaked to add the potential for side effects S:

(`+` Rule)
$$\frac{e_1 \stackrel{S}{\Longrightarrow} v_1, \quad e_2 \stackrel{S'}{\Longrightarrow} v_2 \text{ where } v_1, v_2 \in \mathbb{Z}}{e_1 \texttt{ + } e_2 \stackrel{S \cup S'}{\Longrightarrow} \text{the integer sum of } v_1 \text{ and } v_2}$$

Since e or e' above could in theory have each created actors or sent messages, we need to append their effects to the final result. These effects are like state, they are on the side. A major difference with $\mathbf{F\flat S}$ is the effects here are "write only" – they don't change the direction of local computation in any way, they are only spit out. In that sense local actor computation stays functional.

Here is the send rule:

(`Send` Rule)
$$\frac{e_1 \stackrel{S}{\Longrightarrow} a, \quad e_2 \stackrel{S'}{\Longrightarrow} v}{e_1 \leftarrow e_2 \stackrel{S \cup S' \cup \{[a \leftarrow v]\}}{\Longrightarrow} v}$$

The main consequence is the message $[a \leftarrow v]$ is added to the soup. (The return result v here is largely irrelevant, the goal of a message send is to add the side effect.)

Lastly, here is the create rule:

(`Create` Rule)
$$\frac{e_1 \stackrel{S}{\Longrightarrow} v_1, \quad e_2 \stackrel{S'}{\Longrightarrow} v_2, \quad v_1\ a\ v_2 \stackrel{S''}{\Longrightarrow} v_3}{\texttt{Create}(e_1, e_2) \stackrel{S \cup S' \cup S'' \cup \{\langle a, v_3 \rangle\}}{\Longrightarrow} a, \text{ for } a \text{ a fresh actor name}}$$

This time the return result matters - it is the name of the new actor. The running of $v_1\ a\ v_2$ passes the actor its own name and its initial values to initialize it, and so v_1 will need to be a curried function of two arguments to accept these parameters a and v_2 (in fact it needs to be a curried function of three arguments, because later the message will also be passed as a parameter; more on that very soon).

7.2.5 The Global Rule

Last but not least, here is the global single-step rule for one actor in the soup processing in its entirety one message:

$$(G \cup \{[a \leftarrow v']\} \cup \{\langle a, v \rangle\}) \rightarrow (G \cup \{\langle a, v'' \rangle\} \cup S) \text{ if } (v\ v' \stackrel{S}{\Longrightarrow} v'')$$

This is the only global rule. It matches an actor with a message in the global soup that is destined for it, uses the local semantics to run that actor (in isolation), and throws back into the soup all of the S, which contains all the messages sent by this one actor run as well as any new actors created by this one actor run. A global actor run is just the repeated application of this rule. Notice how the actor behavior which was v is changed to v'', the result of this run.

To test these rules you can run the example programs above.

7.2.6 The Atomicity of Actors

The above semantics has actors executing atomically: each actor runs independently to completion. However, it would be possible to make an alternative semantics in which multiple actors run in parallel Since the actors are completely local in their executions, the two semantics should be provably equivalent.

Chapter 8

Compilation by Program Transformation

The goal of this chapter is to understand the core concepts behind compilation by writing a **F♭SR** compiler. Compilers are an important technology because code produced by a compiler is faster than interpreted code by several orders of magnitude. At least 95% of the production software running is compiled code. Compilation today is a very complex process: compilers make multiple passes on a program to get source code to target code, and perform many complex optimizing transformations. Our goals in this chapter are to understand the most basic concepts behind compilation: how a high-level program can be mapped to machine code.

We will outline a compiler of **F♭SR** to a very limited subset of C ("*pseudo-assembly*"). The reader should be able to implement this compiler in Caml by filling in the holes we have left out. The compiler uses a series of *program transformations* to express the compilation process. These program transformations map **F♭SR** programs to equivalent **F♭SR** programs, removing high-level features one at a time. In particular the following transformations are performed in turn on a **F♭SR** program by our compiler:

1. Closure conversion
2. A-translation
3. Function hoisting

After a program has gone through these transformations, we have a **F♭SR** program that is getting close to the structure of machine language. The last step is then the translate of this primitive **F♭SR** program to C.

Real production compilers such as `gcc` and Sun's `javac` do not use a transformation process, primarily because the speed of the compilation itself is too slow. It is in fact possible to produce very good code by transformation. The SML/NJ ML compiler uses a transformational approach [5]. Also, most production compilers transform the program to an intermediate form which is neither source nor target language ("*intermediate language*") and do numerous optimizing transformations on this intermediate code. Several textbooks cover compiler technology in detail [6, 4].

Our main goal, unlike a production compiler, is *understanding*: to appreciate the gap between high- and low-level code, and how the gaps may be bridged. Each transformation that we define bridges one gap. Program transformations are interesting in their own right, as they give insights into the **F♭SR** language. Optimization, although a central topic in compilation, is beyond the scope of this book. Our focus is on the compilation of higher-order languages, not C/C++; some of the issues are the same but others are different. Also, our executables will not try to catch run-time type errors or garbage collect unused memory.

The desired **soundness property** for each **F♭SR** program translation is: programs before and after translation have the same execution behavior (in our case, termination and same numerical output, but in general the same I/O behavior). Note that the programs that are output by the translation are not necessarily operationally equivalent to the originals.

The **F♭SR** transformations are now covered in the order they are applied to the source program.

8.1 Closure Conversion

Closure conversion is a transformation which eliminates nonlocal variables in functions. For example, `x` in `Function y -> x * y` is a **nonlocal variable**: it is not the parameter and is used in the body. Via closure conversion, all such nonlocal variables can be removed, obtaining an equivalent program where all variables used in functions are parameters of the function. C and C++ have global variables (as does Java via static fields), but global variables are not problematic nonlocal variables. The problematic ones which must be removed are those that are parameters to other functions, where function definitions have been *nested*. C and C++ have no such problematic nonlocal variables since function definitions cannot be nested. In Java, inner classes are nested class definitions, and there is also an issue of nonlocal variables which must be addressed in Java compilation.

Consider for example the following curried addition function.

```
add = Function x -> Function y -> x + y
```

In the body `x + y` of the inner `Function y`, `x` is a nonlocal and `y` is a local variable for that function.

Now, we ask the question, what should `add 3` return? Let us consider some obvious choices:

- `Function y -> x + y` wouldn't make sense because the variable `x` would be undefined, we don't know its value is `3`.
- `Function y -> 3 + y` seems like the right thing, but it amounts to code substitution, something a compiler can't do since compiled code must be immutable.

Since neither of these ideas work, something new is needed. The solution is to return a *closure*, a pair consisting of the function and an *environment* which remembers the values of any nonlocal variables for later use:

```
(Function y -> x + y, { x |-> 3 })
```

Function definitions are now closure definitions; to invoke such a function a new process is needed. **Closure conversion** is a global program transformation that explicitly performs this operation in the language itself. Function values are defined to be *closures*, i.e. tuples of the function and an environment remembering the values of nonlocal variables. When invoking a function which is defined as a closure, we must explicitly pass it the nonlocals environment which is in the closure so it can be used find values of the nonlocals.

The translation is introduced by way of example. Consider the inner `Function y -> x + y` in `add` above translates to the closure

```
{ fn = Function yy -> (yy.envt.x) + (yy.arg);
  envt = { x = xx.arg } };
```

Let us look at the details of the translation. Closures are defined as tuples in the form of records

```
{ fn = Function ...; envt = {x = ...; ...}}
```

consisting of the original function (the `fn` field) and the nonlocals environment (the `envt` field), which is itself a record. In the nonlocals environment `{ x = xx.arg }`, `x` was a nonlocal variable in the original function, and its value is remembered in this record using a *label* of the same name, `x`. All such nonlocal variables are placed in the environment; in this example `x` is the only nonlocal variable.

Functions that used to take an argument `y` are modified to take an argument named `yy` (the original variable name, doubled up). We don't really have to change the name but it helps in understanding because the role of the variable has changed: the new argument `yy` is expected to be a record of the form `{ envt = ..; arg = ..}`, passing both the environment and the original argument to the function.

If `yy` is indeed such a record at function invocation, then within the body we can use `yy.envt.x` to access what was a nonlocal variable `x` in the original function body, and `yy.arg` to access what was the argument `y` to the function.

The whole `add` function is closure-converted by converting both functions:

```
add' = {
   fn = Function xx -> {
      fn = Function yy -> (yy.envt.x) + (yy.arg);
      envt = { x = xx.arg }
   };
   envt = {}
 }
```

The outer `Function x -> ...` arguably didn't need to be closure-converted since it had no nonlocals, but for uniformity it is best to closure convert all functions.

Translation of function application In the above example, there were no applications, and so we didn't define how closure-converted functions are to be applied. Application must change, because functions are now in fact represented as records. Function call `add 3` after closure conversion then must *pass in the environment* since the caller needs to know it:

```
(add'.fn)({ envt = add'.envt; arg = 3})
```

So, we first pull out the function part of the closure, `(add'.fn)`, and then pass it a record consisting of the environment `add'.envt` *also* pulled from the closure, and the argument, `3`. Translation of `add 3 4` takes the result of the above, which should evaluate to a function closure { `fn = ...; envt = ...`}, and does the same trick to apply 4 to it:

```
Let add3' = (add'.fn){ envt = add'.envt; arg = 3 } In
    (add3'.fn){ envt = add3'.envt; arg = 4}
```

and the result would be `12`, the same as the original result, confirming the soundness of the translation in this case. In general applications are converted as follows. At function call time, the remembered environment in the closure is passed to the function in the closure. Thus, for the `add' 3` closure above, `add3'`, when it is applied later to e.g. `7`, the `envt` will know it is `3` that is to be added to `7`.

One more level of nesting Closure conversion is even slightly more complicated if we consider one more level of nesting of function definitions, for example

```
triadd = Function x -> Function y -> Function z -> x + y + z
```

The `Function z` needs to get `x`, and since that `Function z` is defined inside `Function y`, `Function y` has to be an intermediary to pass from the outermost function `x`. Here is the translation.

```
triadd' = {
   fn = Function xx -> {
      fn = Function yy -> {
         fn = Function zz ->
            (zz.envt.x) + (zz.envt.x) + (zz.arg);
         envt = { x = yy.envt.x; y = yy.arg }
      };
      envt = { x = xx.arg }
   };
   envt = {}
}
```

Some observations can be made. The inner `z` function has nonlocals `x` and `y` so both of them need to be in its environment; The `y` function doesn't directly use nonlocals, but it has nonlocal `x` because the function inside it, `Function z`, needs `x`. So its nonlocals `envt` has `x` in it. `Function z` can get `x` into its environment from y's environment, as `yy.envt.x`. Thus, `Function y` serves as middleman to get `x` to `Function z`.

8.1.1 The Official Closure Conversion

With the previous example in mind, we can write out the official closure conversion translation. We will use the notation `clconv(`e`)` to express the closure conversion function, defined inductively as follows (this code is informal; it uses concrete **F♭SR** syntax which in the case of e.g. records looks like Caml syntax).

Definition 8.1 (Closure Conversion)**.**

1. `clconv(x) = x` (* variables *)
2. `clconv(n) = n` (* numbers *)
3. `clconv(b) = b` (* booleans *)
4. `clconv(Function x -> e) =` letting `x, x1, ..., xn` be precisely the free variables in `e`, the result is the **F♭SR** expression

   ```
   { fn = Function xx -> SUB[clconv(e)];
      envt = { x1 = x1; ...; xn = xn } }
   ```

   where `SUB[clconv(e)]` is `clconv(e)` with substitutions `(xx.envt.x1)/x1`, ..., `(xx.envt.xn)/xn` and `(xx.arg)/x` performed on it, but *not* substituting in `Function`'s inside `clconv(e)` (stop substituting when you hit a `Function`).
5. `clconv(e e') = Let f = clconv(e) In (f.fn){ envt = f.envt; arg = clconv(e')}`
6. `clconv(e op e') = clconv(e) op clconv(e')` for all other operators in the language (the translation is *homomorphic* in all of the other operators). This is pretty clear in every case except maybe records which we will give just to be sure...

7. `clconv({ l1 = e1; ...; ln = en }) = { l1 = clconv(e1);...; ln = clconv(en) }`

For the above example, `clconv(add)` is `add'`. The desired soundness result is

Theorem 8.1. *Expression e computes to a value if and only if* `clconv(e)` *computes to a value. Additionally, if one returns numerical value n, the other returns the same numerical value n.*

Closure conversion produces programs where functions have no nonlocal variables, and all functions thus could have been defined at the "top level" like in C. In fact, in Section 8.3 below we will explicitly *hoist* all inner function definitions out to the top.

8.2 A-Translation

Machine language programs are linear sequences of atomic instructions; at most one arithmetic operation is possible per instruction, so many instructions are needed to evaluate complex arithmetic (and other) expressions. The A-translation closes the gap between expression-based programs and linear, atomic instructions, by rephrasing expression-based programs as a sequence of atomic operations. We represent this as a sequence of `Let` statements, each of which performs one atomic operation.

The idea should be self-evident from the case of arithmetic expressions. Consider for instance

```
4 + (2 * (3 + 2))
```

Our **F♭SR** interpreter defined a tree-notion of evaluation order on such expressions. The order in which evaluation happens on this program can be made explicitly linear by using `Let` to factor out the parts in the order that the interpreter evaluates the program

```
Let v1 = 3 + 2 In
Let v2 = 2 * v1 In
Let v3 = 4 + v2 In
    v3
```

This program should give the same result as the original since all we did was to make the computation sequence more self-evident. Notice how similar this is to 3-address machine code: it is a linear sequence of atomic operations directly applied to variables or constants. The `v1` etc variables are *temporaries*; in machine code they generally end up being assigned to registers. These temporaries are not re-used (re-assigned to) above. Register-like programming is not possible in **F♭SR** but it is how real 3-address intermediate language works. In the final machine code generation temporaries are re-used (via a *register allocation* strategy).

We are in fact going to use a more naive (but uniform) translation, that also first assigns constants and variables to other variables:

```
Let v1 = 4 In
Let v2 = 2 In
Let v3 = 3 In
Let v4 = 2 In
Let v5 = v3 + v4 In
Let v6 = v2 * v5 In
Let v7 = v1 + v6 In
    v7
```

This simple translation closely corresponds to the operational semantics—every node in a derivation of `e` $\Rightarrow$ `v` is a `Let` in the above.

Exercise 8.1. Write out the operational semantics derivation and compare its structure to the above.

This translation has the advantage that every operation will be between variables. In the previous example above, `4+v2` may not be low-level enough for some machine languages since there may be no add immediate instruction. One simple optimization would be to avoid making fresh variables for constants. Our emphasis at this point is on correctness as opposed to efficiency, however. `Let` is a primitive in **F♭SR**—this is not strictly necessary, but if `Let` were defined in terms of application, the A-translation results would be harder to manipulate.

Next consider **F♭SR** code that uses higher-order functions.

```
((Function x -> Function y -> y)(4))(2)
```

The function to which `2` is being applied first needs to be computed. We can make this explicit via `Let` as well:

```
Let v1 = (Function x -> Function y -> y)(4) In
Let v2 = v1(2) In
    v2
```

The full A-translation will, as with the arithmetic example, do a full linearization of all operations:

```
Let v1 =
   (Function x ->
      Let v1' = (Function y -> Let v1'' = y in v1'') In v1')
   In
Let v2 = 4 In
Let v3 = v1 v2 In
Let v4 = 2 In
Let v5 = v3 v4 In
    v5
```

All forms of **F♭SR** expression can be linearized in similar fashion, *except* `If`:

```
If (3 = x + 2) Then 3 Else 2 * x
```

can be transformed into something like

```
Let v1 = x + 2 In
Let v2 = (3 = v1) In
If v2 Then 3 Else Let v1 = 2 * x In v1
```

but the `If` still has a branch in it which cannot be linearized. Branches in machine code can be linearized via labels and jumps, a form of expression lacking in **F♭SR**. The above transformed example is still "close enough" to machine code: we can implement it as

```
v1 := x + 2
v2 := 3 = v1
BRANCH v2, L2
L1:  v3 := 3
GOTO L3
L2:  v4 := 4
L3:
```

8.2.1 The Official A-Translation

We define the A-translation as a Caml function, `atrans(e) :  term -> term`. We will always apply A-translation to the result of closure conversion, but that fact is irrelevant for now.

The intermediate result of A-translation is a list of tuples

```
[(v1,e1); ...; (vn,en)] :  (ide * term) list
```

which is intended to represent

```
Let v1 = e1 In ... In Let vn = en In vn ...
```

but is a form easier to manipulate in Caml since lists of declarations will be appended together at translation time. When writing a compiler, the programmer may or may not want to use this intermediate form. It is not much harder to write the functions to work directly on the `Let` representation.

We now sketch the translation for the core primitives. Assume the following auxiliary functions have been defined:

- `newid()` which returns a fresh **F♭SR** variable every time called,
- The function `letize` which converts from the list-of-tuples form to the actual `Let` form, and
- `resultId`, which for list `[(v1,e1); ...; (vn,en)]` returns result identifier `vn`.

Definition 8.2 (A Translation)**.**

```
let atrans e = letize (atrans0 e)

and atrans0(e) = match e with
  (Var x) -> [(newid(),Var x)]
| (Int n) -> [(newid(),Int n)]
| (Bool b) -> [(newid(),Bool b)]
| Function(x,e) -> [(newid(),Function(x,atrans e)]
| Appl(e,e') -> let a = atrans0 e in let a' = atrans0 e' in
    a @ a' @ [(newid(),Appl(resultId a,resultId a'))]

(* all other D binary operators + - = AND etc. of form
 * identical to Appl
 *)

| If(e1,e2,e3) -> let a1 = atrans0 e1 in
    a1 @ [(newid();If(resultId a1,atrans e2,atrans e3)]

(* ... *)
```

At the end of the A-translation, the code is all "linear" in the way it runs in the interpreter, not as a tree. Machine code is also linearly ordered; we are getting much closer to machine code.

Theorem 8.2. *A-translation is sound, i.e.* `e` *and* `atrans(e)` *both either compute to values or both diverge.*

Although we have only partially defined A-translation, the extra syntax of **F♭SR** (records, reference cells) does not provide any major complication.

8.3 Function Hoisting

So far, we have defined a front end of a compiler which performs closure conversion and A-translation in turn:

```
let atrans_clconv e = atrans(clconv(e))
```

After these two phases, functions will have no nonlocal variables. Thus, we can *hoist* all functions in the program body to the start of the program. This brings the program

structure more in line with C (and machine) code. Since our final target is C, the leftover code from which all functions were hoisted then is made the `main` function. A function `hoist` carries out this transformation. Informally, the operation is quite simple: take e.g.

```
4 + (Function x -> x + 1)(4)
```

and replace it by

```
Let f1 = Function x -> x + 1 In 4 + f1(4)
```

In general, we hoist all functions to the front of the code and give them a name via `Let`. The transformation is always sound if there are no free variables in the function body, a property guaranteed by closure conversion. We will define this process in a simple iterative (but inefficient) manner:

Definition 8.3 (Function Hoisting)**.**

```
let hoist e =
   if e = e1[(Function ea -> e')/f] for some e1 with f free,
        and e' itself contains no functions
       (i.e. Function ea -> e' is an innermost function)
   then
      Let f = (Function ea -> e') In hoist(e1)
   else e
```

This function hoists out innermost functions first. If functions are not hoisted out innermost-first, there will still be some nested functions in the hoisted definitions. So, the order of hoisting is important.

The definition of hoisting given above is concise, but it is too inefficient. A one-pass implementation can be used that recursively replaces functions with variables and accumulates them in a list. This implementation is left as an exercise. Resulting programs will be of the form

```
Let f1 = Function x1 -> e1 In
...
Let fn = Function xn -> en In
e
```

where each $e, e_1, \ldots e_n$ contain no function constants.

Theorem 8.3. *If all functions occurring in expression* `e` *contain no nonlocal variables, then* `e` $\cong$ `hoist(e)`.

This Theorem may be proved by iterative application of the following Lemma:

Lemma 8.1.

$e_1[($`Function` x `->` $e')/f] \cong$
(`Let` f `= (Function` x `->` e'`) In` e_1

provided e' *contains at most the variable* x *free.*

We lastly transform the program to

```
Let f1 = Function x1 -> e1 In
    ...
      fn = Function -> Function xn -> en In
      main = Function dummy -> e In
      main(0)
```

So, the program is almost nothing but a collection of functions, with a body that just invokes `main`. This brings the program closer to C programs, which are nothing but a collection of functions and `main` is implicitly invoked at program start.

`Let Rec` definitions also need to be hoisted to the top level; their treatment is similar and will be left as an exercise.

8.4 Translation to C

We are now ready to translate into C. To summarize up to now, we have

```
let hoist_atrans_clconv e = hoist(atrans(clconv(e)))
```

We have done about all the translation that is possible within **F♭SR**. Programs are indeed looking a lot more like machine code: all functions are declared at the top level, and each function body consists of a linear sequence of atomic instructions (with exception of `If` which is a branch). There still are a few things that are more complex than machine code: records are still implicitly allocated, and function call is atomic, no pushing of parameters is needed. Since C has function call built in, records are the only significant gap that needs to be closed.

The translation involves two main operations.

1. Map each function to a C function
2. For each function body, map each atomic tuple to a primitive C statement.

Atomic Tuples Before giving the translation, we enumerate all possible right-hand sides of `Let` variable assignments that come out of the A-translation (in the following `vi, vj, vk`, and `f` are variables).These are called the **atomic tuples**.

Fact 8.1 (Atomic Tuples)**.** ***FVSR*** *programs that have passed through the first three phases have function bodies consisting of tuple lists where each tuple is of one of the following forms only:*

1. `x` for variable `x`
2. `n` for number `n`
3. `b` for boolean `b`
4. `vi vj` (application)
5. `vj + vk`
6. `vj - vk`
7. `vj And vk`
8. `vj Or vk`
9. `Not vj`
10. `vj = vk`
11. `Ref vj`
12. `vj := vk`
13. `!vj`
14. `{ l1 = v1; ...; ln = vn }`
15. `vi.l`
16. `If vi Then tuples1 Else tuples2` where `tuples1` and `tuples2` are the lists of variable assignments for the `Then` and `Else` bodies.

Functions should have all been hoisted to the top so there will be none of those in the tuples. Observe that some of the records usages are from the original program, and others were added by the closure conversion process. We can view all of them as regular records. All we need to do now is generate code for each of the above tuples.

8.4.1 Memory Layout

Before writing any compiler, a fixed memory layout scheme for objects at run-time is needed. Since objects can be read and written from many different program points, if read and write protocols are not uniform for a given object, the code simply will not work! So, it is very important to carefully design the strategy beforehand. Simple compilers such as ours will use simple schemes, but for efficiency it is better to use a more complex scheme.

Lets consider briefly how memory is laid out in C. Values can be stored in several different ways:

- In registers: These must be temporary, as registers are generally local to each function/method. Also, registers are only one word in size (or a couple words, for floats) so can't directly hold arrays or structs.
- On the run-time stack in the function's *activation record*. The value is then referenced as the memory location at some fixed offset from the stack pointer (which is itself in a register).
- In fixed memory locations (globals).
- In dynamically allocated (`malloc`'ed) memory locations (on the heap).

Figure 8.1 illustrates the overall model of memory we are dealing with.

To elaborate a bit more on stack storage of variables, here is some C pseudo-code to give you the idea of how the stack pointer `sp` is used.

```
register int sp; /* compiler assigns sp to a register */
*(sp - 5) = 33;
printf("%d", *(sp - 5));
```

Stack-stored entities are also temporary in that they will be junk when the function/method returns.

Another important issue is whether to **box** or **unbox** values.

Definition 8.4. *A register or memory location v_i's value is stored* boxed *if v_i holds a pointer to a block of memory containing the actual value. A variable's value is* unboxed *if it is directly in the register or memory location v_1.*

Figure 8.2 illustrates the difference between boxed and unboxed values.

For multi-word entities such as arrays, storing them unboxed means variables directly hold a pointer to the first word of the sequence of space. To clarify the above concepts we review C's memory layout convention. Variables may be declared either as globals, register (the register directive is a request to put in a register only), or on the call stack; all variables declared inside a function are kept on the stack. Variables directly holding `int`s, `float`s, `struct`s, and arrays are all unboxed. (Examples: `int x; float x; int arr[10]; snork x` for `snork` a `struct`.) There is no such thing as a variable directly holding a function; variables in C may only hold *pointers* to functions. It is possible to write "`v = f`" in C where `f` is a previously declared function and not "`v = &f`", but that

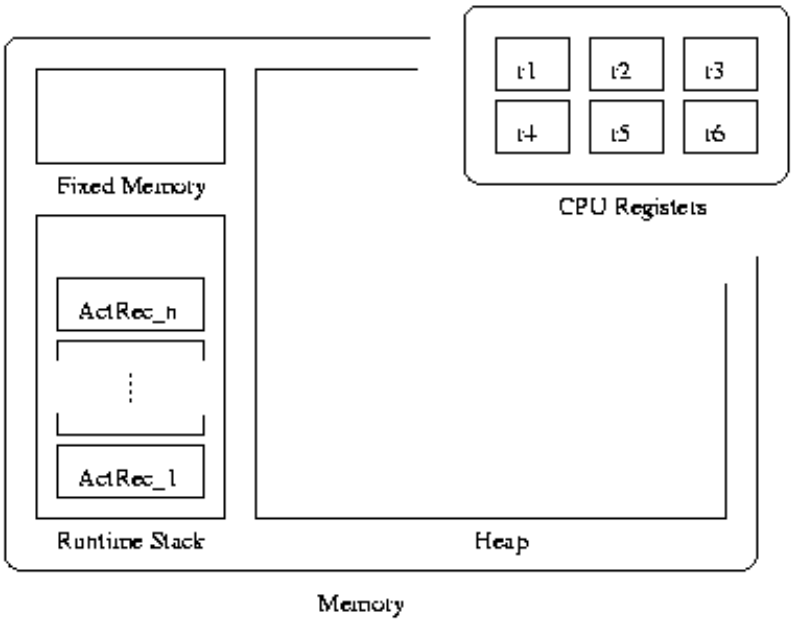


Figure 8.1: Our model of memory

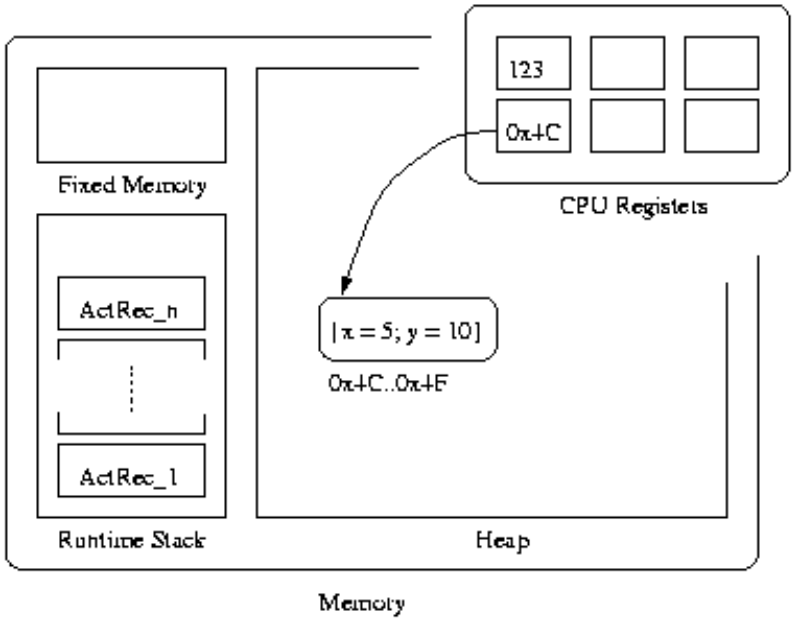


Figure 8.2: Boxed vs. unboxed values. The integer value `123` is stored as an unboxed value, while the record {`x=5; y=10`} is stored as a boxed value.

is because the former is really syntactic sugar for the latter. A pointer to a function is in fact a pointer to the start of the code of the function. Boxed variables in C are declared explicitly, as pointer variables. (Examples: `int *x; float *x; int *arr[10]; snork *x` for `snork` a `struct`.) All `malloc`'ed structures must be stored in a pointer variable because they are boxed: variables can't directly be heap entities. Variables are static and the heap is dynamic.

Here is an example of a simple C program and the Sun SPARC assembly output which gives some impressionistic idea of these concepts:

```
int glob;
main()
{
   int x;
   register int reg;
   int* mall;
   int arr[10];

   x = glob + 1;
   reg = x;
   mall = (int *) malloc(1);
   x = *mall;
   arr[2] = 4;
/* arr = arr2; --illegal:  arrays are not boxed */
}
```

In the assembly language, `%o1` is a register, `[%o0]` means dereference, `[%fp-24]` means subtract 24 from frame pointer register `%fp` and dereference. The assembly representation of the above C code is as follows.

```
main:
     sethi   %hi(glob), %o1
     or      %o1, %lo(glob), %o0 /* load global address glob into %o0 */
     ld      [%o0], %o1    /* dereference */
     add     %o1, 1, %o0   /* increment */
     st      %o0, [%fp-20] /* store in [%fp-20], 20 back from fp -- x */
                           /* x directly contains a number,
                           /* not a pointer */
     ld      [%fp-20], %l0 /* %l0 IS reg (its in a register directly) */
     mov     1, %o0
     call    malloc, 0     /* call malloc.  resulting address to %o0 */
     nop
     st      %o0, [%fp-24] /* put newspace location in mall [%fp-24] */
     ld      [%fp-24], %o0 /* load mall into %o0 */
     ld      [%o0], %o1    /* this is a malloced structure -- unbox. */
     st      %o1, [%fp-20] /* store into x */
     mov     4, %o0
     st      %o0, [%fp-56] /* array is a sequence of memory on stack  */
.LL2:
```

```
ret
restore
```

Our memory layout strategy is more like a higher-level language such as Java or ML. The Java JVM uses a particular, fixed, memory layout scheme: all object references are boxed pointers to heap locations; the primitive `bool`, `byte`, `char`, `short`, `int`, `long`, `float`, and `double` types are kept unboxed. Since Java arrays are objects they are also kept boxed. There is no reference (`&`) or dereference (`*`) operator in Java. These operations occur implicitly.

Memory layout for our F♭SR compiler **F♭SR** is (mostly) a Caml subset and so its memory is also managed more implicitly than C memory. We will use a simple, uniform scheme in our compilers which is close in spirit to Java's: Box `Ref`s and records and function values, but keep boolean values and integers unboxed. Also, as in C (and Java), all local function variables will be allocated on the stack and accessed as offsets from the stack pointer. We will achieve the latter by implementing **F♭SR** local variables as C local variables, which will be stack allocated by the C compiler.

Since a `Ref` is nothing but a mutable location, there may not seem to be any reason to box it. However, if a function returns a `Ref` as result, and it were not boxed, it would have been allocated on the stack and thus would be deallocated. Here is an example that reflects this problem:

```
Let f = (Function x -> Ref 5) In !f(_) + 1
```

If `Ref 5` were stored on the stack, after the return it could be wiped out. All of the `Let`-defined entities in our tuples (the `vi` variables) can be either in registers or on the call stack: none of those variables are directly used outside the function due to lexical scoping, and they don't directly contain values that should stay alive after the function returns. For efficiency, they can all be declared as `register Word` variables:

```
register Word v1, v2, v3, v4, ...;
```

One other advantage of this simple scheme is every variable holds one word of data, and thus we don't need to keep track of how much data a variable is holding. This scheme is not very efficient, and real compilers optimize significantly. One example is `Ref`'s which are known to not escape a function can unboxed and stack allocated.

All that remains is to come up with a scheme to compile each of the above atomic tuples and we are done. Records are the most difficult so we will consider them before writing out the full translation.

Compiling untyped records Recall from when we covered records that the fields present in a record cannot be known in advance if there is no type system. So, we won't know where the field that we need is exactly. Consider, for example,

```
(Function x -> x.l)(If y = 0 Then {l = 3} Else {a = 4; l = 3})
```

Field `l` will be in two different positions in these records so the selection will not have a sole place it can find the field in. Thus we will need to use a hashtable for record lookup. In a typed language such as Caml this problem is avoided: the above code is not well-typed in Caml because the if-then can't be typed. Note that the problems with records are closely related to problems with objects, since objects are simply records with `Ref`s.

This memory layout difficulty with records illustrates an important relationship between typing and compilation. Type systems impose constraints on program structure that can make compilers easier to implement. Additionally, typecheckers will obviate the need to deal with certain run-time errors. Our simple **F♭SR** compilers are going to core dump on e.g. `4 (5)`; in Lisp, Smalltalk, or Scheme these errors would be caught at run-time but would slow down execution. In a typed language, the compiler would reject the program since it will not typecheck. Thus for typed languages they will both be faster and safer.

Our method for compilation of records proceeds as follows. We must give records a heavy implementation, as hash tables (i.e., a set of key-value pairs, where the keys are label names). In order to make the implementation simple, records are boxed so they take one word of memory, as mentioned above when we covered boxing. A record selection operation $v_k.l$ is implemented by hashing on key l in the hash table pointed to by v_k at runtime. This is more or less how Smalltalk message sends are implemented, since records are similar to objects (and Smalltalk is untyped).

The above is less than optimal because space will be needed for the hashtable, and record field accessing will be *much* slower than, for example, `struct` access in C. Since closures are records, this will also significantly slow down function call. A simple optimization would be to treat closure records specially since the field positions will always be fixed, and use a `struct` implementation of closure (create a different `struct` type for each function).

For instance, consider

```
(Function x -> x.l)(If y = 0 Then {l = 3} Else {a = 4; l = 3})
```

The code `x.l` will invoke a call of approximate form `hashlookup(x,"l")`. {`a = 4; l = 3`} will create a new hash table and add mappings of `"a"` to 4 and `"l"` to 3.

8.4.2 The toC translation

We are now ready to write the final translation to C, via functions

- `toCTuple` mapping an atomic tuple to a C statement string,
- `toCTuples` mapping a list of tuples to C statements,
- `toCFunction` mapping a primitive **F♭SR** function to a string defining a C function,
- `toC` mapping a list of primitive **F♭SR** functions to a string of C functions.

The translation as informally written below takes a few liberties for simplicity. Strings `"..."` below are written in shorthand. For instance `"vi = x"` is shorthand for `tostring(vi) ^" = " ^tostring(x)`. The tuples `Let x1 = e1 In Let ...In Let xn = en In xn` of function and then/else bodies are assumed to have been converted to lists of tuples `[(x1,e1),...,(xn,en)]`, and similarly for the list of top-level function definitions. When writing a compiler, it probably will be easier just to simply keep them in `Let` form, although either strategy will work.

```
toCTuple(vi = x) =           "vi = x;" (* x is a FbSR variable *)
toCTuple(vi = n) =           "vi = n;"
toCTuple(vi = b) =           "vi = b;"
toCTuple(vi = vj + vk) =     "vi = vj + vk;"
toCTuple(vi = vj - vk) =     "vi = vj - vk;"
toCTuple(vi = vj And vk ) =  "vi = vj && vk;"
toCTuple(vi = vj Or vk ) =   "vi = vj || vk;"
toCTuple(vi = Not vj ) =     "vi = !vj;"
toCTuple(vi = vj = vk) =     "vi = (vj == vk);"
toCTuple(vi = (vj vk) =      "vi = *vj(vk);"
toCTuple(vi = Ref vj) =      "vi = malloc(WORDSIZE); *vi = vj;"
toCTuple(vi = vj := vk) =    "vi = *vj = vk;"
toCTuple(vi = !vj) =         "vi = *vj;"
toCTuple(vi = { l1 = v1; ... ; ln = vn }) =
         /* 1. malloc a new hashtable at vi
            2. add mappings l1 -> v1 , ... , ln -> vn  */

toCTuple(vi = vj.l) =        "vi = hashlookup(vj,"l");"
toCTuple(vi = If vj Then tuples1 Else tuples2) =
  "if (vj) { toCTuples(tuples1) } else { toCTuples(tuples2) };"
toCtuples([]) = ""

toCtuples(tuple::tuples) = toCtuple(tuple) ^ toCtuples(tuples)

toCFunction(f = Function xx -> tuples) =
  "Word f(Word xx) {" ^ ... declare temporaries ...
  toCtuples(tuples) ^
 "return(resultId tuples); };"

toCFunctions([]) = ""
toCFunctions(Functiontuple::Functiontuples) =
  toCFunction(Functiontuple) ^ toCFunctions(Functiontuples)

(* toC then invokes toCFunctions on its list of functions. *)
```

The reader may wonder why a fresh memory location is allocated for a `Ref`, as opposed to simply storing the existing address of the object being referenced. This is a subtle issue, but the code `vi = &vj`, for example, would definitely not work for the `Ref` case (`vj` may go out of scope).

This translation sketch above leaves out many details. Here is some elaboration.

Typing issues We designed out memory layout so that every entity takes up one word. So, every variable is of some type that is one word in size. Type all variables

as `Word`'s, where `Word` is a 1-word type (defined as e.g. `typedef void *Word;`). Many type casts need to be inserted; we are basically turning off the type-checking of C, but there is no "switch" that can be flicked. So, for instance `vi = vj + vk` will really be `vi = (Word (int vj) + (int vk))` – cast the words to `int`s, do the addition, and cast back to a word. To cast to a function pointer is a tongue-twister: in C you can use `(*((Word (*)()) f))(arg)`. The simplest way to avoid confusion when actually writing a compiler is to include the following `typedef`s to the resulting C code:

```
/*
 * Define the type 'Word' to be a generic one-word value.
 */
typedef void *Word;

/*
 * Define the type 'FPtr' to be a pointer to a function that
 * consumes Word and returns a Word.  All translated
 * functions should be of this form.
 */
typedef Word (*FPtr)(Word);

/*
 * Here is an example of how to use these typedefs in code.
 */
Word my_function(Word w) {
  return w;
}
int main(int argc, char **argv) {
  Word f1 = (Word) my_function;
  Word w1 = (Word) 123;
  Word w2 = ( *((FPtr) f1) )(w1); /* Computes f1(123) */
  printf("%d\n", (int) w2);       /* output is "123\n". */
  return 0;
}
```

Global Issues Some global issues you will need to deal with include the following. You will need to print out the result returned by the `main` function (so, you probably want the FbSR main function to be called something like `FbSRmain` and then write your own `main()` by hand which will call `FbSRmain`); The C functions need to declare all the temporary variables they use. One solution is to declare in the function header a C array

```
Word v[22]
```

where 22 is the number of temporaries needed in this particular function, and use names `v[0], v[1],` etc for the temporaries. Note, this works well only if the `newid()` function

is instructed to start numbering temporaries at zero again upon compiling each new function. Every compiled program is going to have to come with a standard block of C code in the header, including the record hash implementation, `main()` as alluded to above, etc.

Other issues that present problems include the following. Record creation is only sketched; but there are many C hash set libraries that could be used for this purpose. The final result (`resultId`) of the `Then` and `Else` tuples needs to be in the *same* variable `vi`, which is also the variable where the result of the tuple is put, for the `If` code to be correct. This is best handled in the A-translation phase.

There are several other issues that will arise when writing the compiler. The full implementation is left as an exercise.

8.4.3 Compilation to Assembly code

Now that we have covered the compilation to C, we informally consider how a compiler would compile all the way to machine code. The C code we produce above is very close to assembly code. It would be conceptually easy to translate into assembly, but we skip the topic due to the large number of cases that arise in the process (saving registers, allocating space on the stack for temporaries).

8.5 Summary

```
let frontend e = hoist(atrans(clconv(e)));;
let translator e = toC(frontend(e));;
```

We can assert the correctness of our translator.

Theorem 8.4. *FbSR program* `e` *terminates in the* ***FbSR*** *operational semantics (or evaluator) just when the C program* `translator(e)` *terminates, provided the C program does not run out of memory. Core dump or other run-time errors are equated with nontermination. Furthermore, if* ***FbSR****'s* `eval(e)` *returns a number* `n`*, the compiled* `translator(e)` *will also produce numerical output* `n`*.*

8.6 Optimization

Optimization can be done at all phases of the translation process. The above translation is simple, but inefficient. There is always a tradeoff between simplicity and efficiency in compiler designs, both the efficiency of compilation itself, and efficiency of code produced. In the phases before C code is produced, optimizations consist of replacing chunks of the program with operationally equivalent chunks.

Some simple optimizations include the following. The special closure records {`fn = .., envt = ..` } could be implemented as a pointer to a C `struct` with `fn` and `envt` fields, instead of using the very slow hash method,[1] will significantly speed up the code

[1] Although hash lookups are $O(1)$, there is still a large amount of constant overhead, whereas `struct` access can be done in a single load operation.

produced. Records which do not not have field names overlapping with other records can also be implemented in this manner (there can be two different records with the same fields, but not two different records with some fields the same and some different). Another optimization is to modify the A-translation to avoid making tuples for variables and constants. Constant expressions such as `3 + 4` can be folded to `7`.

More fancy optimizations require a global **flow analysis** be performed. Simply put, a flow analysis finds all possible *uses* of a particular definition, and all possible *definitions* corresponding to a particular use.

A definition is a record, a `Function`, or a number or boolean, and a use is a record field selection, function application, or numerical or boolean operator.

8.7 Garbage Collection

Our compiled code `mallocs` but never frees. We will eventually run out of memory. A garbage collector is needed.

Definition: In a run-time image, memory location n is *garbage* if it never will be read or written to again.

There are many notions of garbage detection. The most common is to be somewhat more conservative and take garbage to be memory locations which are not pointed to by any known ("root") object.

Bibliography

[1] Open source erlang. http://www.erlang.org/.

[2] The Self programming language. http://research.sun.com/self/language.html.

[3] Typed assembly language. http://www.cs.cornell.edu/talc/.

[4] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[5] A. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[6] Andrew Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[7] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 2nd edition, 1998.

[8] Gibblad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a Production Environment. In *Proceedings of the OOPSLA '93 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 215–230, 1993.

[9] Kim Bruce. *Foundations of Object-Oriented Languages: Types and Semantics*. MIT Press, 2002.

[10] Jonathan Eifrig, Scott Smith, Valery Trifonov, and Amy Zwarico. Application of OOP type theory: State, decidability, integration. In *OOPSLA '94*, pages 16–30, 1994.

[11] Martin Fowler. *UML Distilled*. Addison Wesley, 2nd edition, 2000.

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1994.

[13] Paul Hudak, John Peterson, and Joseph Fasel. A gentle introduction to Haskell, version 98, June 2000. http://www.haskell.org/tutorial/.

[14] Andrew D. Irvine. Russell's paradox. Stanford Encyclopedia of Philosophy, June 2001. http://plato.stanford.edu/entries/russell-paradox/.

[15] Xavier Leroy. The Objective Caml system release 3.11, documentation and user's manual, November 2008. http://caml.inria.fr/pub/docs/manual-ocaml/index.html.

[16] Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. From operational semantics to domain theory. *Information and Computation*, 128(1):26–47, 1996.

[17] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. Talx86: A realistic typed assembly language. In *1999 ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, GA, USA, May 1999.

[18] J J O'Connor and E F Robertson. Gottfried Wilhelm von Leibniz. The MacTutor History of Mathematics Archive, October 1998. http://www-history.mcs.st-andrews.ac.uk/history/Mathematicians/Leibniz.html.

[19] Randall B. Smith and David Ungar. Self: The power of simplicity. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 227–242. ACM Press, 1987.

[20] Randall B. Smith and David Ungar. Programming as an experience: The inspiration for Self. *Lecture Notes in Computer Science*, 952:303–??, 1995.

[21] Scott Smith. Programming languages course. http://pl.cs.jhu.edu/pl.

[22] J. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[23] Paul R. Wilson. Uniprocessor garbage collection techniques. *ACM Computing Surveys*, 2002. ftp://ftp.cs.utexas.edu/pub/garbage/bigsurv.ps.

Index