CS 3366 Programming Languages

Prof. R. Muller

The Programming Language Venus

1 Introduction

Venus is a simple expression-based "programming" language designed with an eye toward exhibiting some important properties of real programming languages in the simplest possible setting. Venus is actually a family of dialects that share common abstract syntax but vary in their approaches to typing and evaluation order.

Like Mercury, Venus is based in the tradition of symbolic systems where program execution is carried out by simplification rules that drive expressions toward their values if they have one. The main additions over and above the simple `int` expressions found in Mercury are

1. a single additional type, `real`, and
2. the ability to **name** things.

The main properties of Venus will be illustrated in a few examples. An example Venus program is

```
1. let x : int = 2 + r2i(3.9) in
2. let y : real = i2r(x)
3. in
4. y *. i2r(x)
```

The code in lines 1. - 4. is a well-typed Venus-program that evaluates to the real numeral `25.0`. In line 1., the symbol `x` is introduced. This is a *binding occurrence* of `x`; it governs the *uses* of `x` on lines 2. and 4. The binding occurrence of `x` includes a type annotation `:int`; the coder is asserting that the variable `x` will hold a value of type `int`. The initialization expression `2 + r2i(3.9)` has an integer literal, a real literal, a type coercion function `r2i` and an integer addition operation.

Typing

The addition of the type `real` is very helpful to the coder but it introduces potential problems. Integers and reals are represented differently on digital computers. The former are in *two's complement* representation while the latter are in the *IEEE-754 Floating Point Standard* representation. So the language designer has to decide whether or not to help the coder keep things straight, and if so, how. For small programs it isn't hard, but as the code base grows large, it becomes an extremely difficult problem.

In the example, the coder's assertions turn out to be correct, a fact that the compiler may or may not confirm, depending on the type discipline. We'll consider four main variations of the typing discipline.

Evaluation Order

The execution of this code might proceed simplifying as follows

```
let x : int = 2 + r2i(3.9) in let y : real = i2r(x) in y *. i2r(x) -->
let x : int = 2 + 3 in let y : real = i2r(x) in y *. i2r(x) -->
let x : int = 5 in let y : real = i2r(x) in y *. i2r(x) -->
let y : real = i2r(5) in y *. i2r(5) -->
let y : real = 5.0 in y *. i2r(5) -->
5.0 *. i2r(5) -->
5.0 *. 5.0 -->
25.0
```

This reflects the *call-by-value* evaluation order. The same code might have been executed, simplifying as follows

```
let x : int = 2 + r2i(3.9) in let y : real = i2r(x) in y *. i2r(x) -->
let y : real = i2r(2 + r2i(3.9)) in y *. i2r(2 + r2i(3.9)) -->
i2r(2 + r2i(3.9)) *. i2r(2 + r2i(3.9)) -->
i2r(2 + 3) *. i2r(2 + r2i(3.9)) -->
i2r(5) *. i2r(2 + r2i(3.9)) -->
5.0 *. i2r(2 + r2i(3.9)) -->
5.0 *. i2r(2 + 3) -->
5.0 *. i2r(5) -->
5.0 *. 5.0 -->
25.0
```

This reflects the *call-by-name* evaluation order. There are important trade-offs between call-by-value and call-by-name. There are other evaluation orders too including *call-by-need* and *parallel simplification*.

Abstract Syntax

In our discussion of the various dialects of Venus, we're going to ignore issues relating to concrete syntax. The abstract syntax is given in Figure 1.

2 Symbolic Names and Scope

One of the most important tasks in coding is inventing *symbolic names* for things. Coders name all sorts of things: values, types, constructors, fields of records, classes, modules, packages, exceptions, applications, groups, companies, etc. The use of informative names is extraordinarily important in the software life cycle. Good coders spend a lot of time thinking

$$
\begin{array}{rcl}
\texttt{Top} & ::= & \texttt{let D} \mid \texttt{E} \\
\texttt{D} & ::= & \texttt{x} = \texttt{E} \mid \texttt{x} : \tau = \texttt{E} \\
\texttt{E} & ::= & \texttt{V} \mid \texttt{x} \mid \texttt{E op E} \mid \texttt{i2r(E)} \mid \texttt{r2i(E)} \mid \texttt{let D in E} \\
\tau & ::= & \texttt{int} \mid \texttt{real} \\
\texttt{V} & ::= & \texttt{i} \mid \texttt{r} \\
\texttt{op} & ::= & + \mid - \mid * \mid / \mid \% \mid +. \mid -. \mid *. \mid /.
\end{array}
$$

Figure 1: The abstract syntax of Venus, `x` is an identifier, `i` is an integer, `r` is a real.

about how to name things in such a way that the reader will have no trouble internalizing the role played by the value of the symbol.

The association between a name and some object is called a **binding** and programming languages come with various sorts of *binding forms* that allow the code to associate names with various things. For example, in Java, the class definition form:

```
public class Vehicle { ... }
```

binds the name `Vehicle` to the definition of the class. This name can be used in certain places to create objects of the class. When a name is introduced, where can it be used? The rules can be fairly tricky, with names sometimes being qualified by package, class or module names.

For the purposes of Venus, we can focus on resolving symbolic names in a simpler setting. Venus has only one binding form, the `let`-expression. This is a simple form but it embodies a lot of the complexity of more complicated forms in production programming languages.

The `let`-expression in Venus has two variations. There is a top-level `let`

```
let x : t = E
```

which enters bindings into the top-level environment, and the standard expression-oriented `let-in` form:

```
let x : t = E1 in E2
```

In both of these, the type annotation `:t` is optional. We're going to focus on the standard `let-in` variation.

The `let-in`-expressions introduces the symbolic name `x` that can be used in `E2`. But it's not the case that every occurrence of `x` in `E2` is a use of the variable bound in the `let` form. For example, one might have

```
let x = 1 in let x = 2 in x + x
```

The `x`s in the body should be `2` rather than `1`. In this case, the binding of `x` to `2` is said to **shadow** the binding of `x` to `1`. In order to define this precisely, we use the notion of *free variables*.

Free and Bound Variables

$$\begin{array}{rcl}
\text{FV}(\texttt{V}) & = & \{\} \\
\text{FV}(\texttt{x}) & = & \{\texttt{x}\} \\
\text{FV}(E_1 \ \texttt{op} \ E_2) & = & \text{FV}(E_1) \cup \text{FV}(E_2) \\
\text{FV}(\texttt{i2r}(E)) & = & \text{FV}(E) \\
\text{FV}(\texttt{r2i}(E)) & = & \text{FV}(E) \\
\text{FV}(\texttt{let x} = E_1 \ \texttt{in} \ E_2) & = & \text{FV}(E_1) \cup (\text{FV}(E_2) - \{\texttt{x}\})
\end{array}$$

A program E such that $\text{FV}(E) = \{\}$ is said to be **closed**. A program that is not closed is said to be **open**. For example, $\text{FV}(\texttt{let x} = 3 \ \texttt{in x}) = \{\}$ and $\text{FV}(\texttt{let x} = \texttt{x in x}) = \{\texttt{x}\}$, the former is closed while the latter is open.

Notes

1. From the definition, we see that in the `let-in` expression, only the free occurrences of `x` in E_2 are bound by the binding occurrence `x`.

2. Note that the "freeness" of an occurrence of a variable is a relative concept. The rightmost `x` in `let x` = 12 `in` x is free with respect to itself but it is bound with respect to the whole expression.

3. A closed program is sometimes called a **combinator**.

Lexical Scoping and Block Structure

Since the `let-in` form `let x` = E_1 `in` E_2 allows nesting in both E_1 and E_2, it gives rise to a scoping discipline known as **block structure** — a term that was coined in reference to Algol-60 in the late 1950s. This means that as far as name resolution is concerned, we can draw a hierarchical collection of blocks. For example, the scope structure of the Venus code:

```
let x1 = (let x2 = 2 in x2 + x2) + (let x3 = 3 in x3 + x3) in
let x4 = x1 in
let x5 = x4
in
x1 + x4 + x5
```

can be understood as the block structure illustrated in Listing 1.

Listing 1: Lexical Block Structure

```
+----------------------------------------------+
| x1 = +----------------+   +----------------+ |
|      | x2 = 2         |   | x3 = 3         | |
|      | in             |   | in             | |
|      | +------------+ | + | +------------+ | |
|      | | x2 + x2    | |   | | x3 + x3    | | |
|      | +------------+ |   | +------------+ | |
|      +----------------+   +----------------+ |
| in                                           |
|  +-----------------------------------------+ |
|  | x4 = x1                                 | |
|  | in                                      | |
|  | +-------------------------------------+ | |
|  | | x5 = x4                             | | |
|  | | in                                  | | |
|  | | +---------------------------------+ | | |
|  | | |          x1 + x4 + x5           | | | |
|  | | +---------------------------------+ | | |
|  | +-------------------------------------+ | |
|  +-----------------------------------------+ |
+----------------------------------------------+
```

The variables occurrences `x1`, `x4` and `x5` are all free in the innermost block. One just scans inside-out in containing blocks to find the appropriate binding occurrence. If no binding occurrence is found, the variable is unbound.

Substitution

Returning to the first example, the simplification of the code eventually arrived at a `let-in` form

```
...
let x : int = 5 in let y : real = i2r(x) in y *. i2r(x) -->
let y : real = i2r(5) in y *. i2r(5) -->
...
```

This simplification step required the *replacement* or *substitution* of an expression for the bound variable `x`. In the example, we plugged `5` in for the two occurrences of `x` in the expression `i2r(x) in y *. i2r(x)`. This yielded the expression `i2r(5) in y *. i2r(5)`.

The basic idea of substitution, although intuitive, is actually a little tricky in the general case and requires a careful definition. Two issues are *shadowing* and *name capture*.

Shadowing

The following example has two different binding occurrences if `x`.

```
let x = 12 in (let x = x * 2 in x + x) + x
```

In addition to the two binding occurrences there are four uses (or *applied occurrences*) of `x`. To simplify the explanation, lets number the occurrences from left to right:

```
let x0 = 12 in (let x1 = x2 * 2 in x3 + x4) + x5
```

The applied occurrences of a variable that are *bound* (or governed) by a given binding occurrence are called the *scope* of the binding occurrence.

In our example, the scope of `x0` is the set of applied occurrences `{x2, x5}` and the scope of the binding occurrence `x1` is the set of applied occurrences `{x3, x4}`.

The example illustrates that one has to be a little careful in plugging in expressions for variables. In particular, the nested `let`-form forms a hole in the scope of `x0` — the binding occurrence `x1` casts a shadow over the occurrences of `x` in the expression `x3 + x4`. If we wish to simplify the expression by plugging `12` in for `x`, we must proceed as follows:

```
let x0 = 12 in (let x1 = x2 * 2 in x3 + x4) + x5
--> (let x1 = 12 * 2 in x3 + x4) + 12
--> ...
```

leaving the variable occurrences `x3` and `x4` unchanged.

Name Capture

A related problem involves the possible *capture* of free applied occurrences of variables by binding occurrences of the same name. For example, if we naively substitute `x * x` for `z` in `let x = 4 in x * z`

```
(let x = 4 in x * z)[z := x * x]
```

we'd be left with

```
(let x = 4 in x * x * x)
```

which computes 4^3. But the two free occurrences of `x` in `x * x` were likely bound to something else, say `2`, in the definition environment so naive substitution leaves us with an **incorrect answer**.

Mistakes of this kind appeared in early dialects of the programming language LISP. At the time this property was thought to be a "feature" and it was called *dynamic scoping*. Today it is generally understood that dynamic scoping is a cause of many hard-to-debug errors and is a serious bug in the design of LISP, though it is still in use in E-LISP and in LaTeX.

The plug-in or replacement process can be defined precisely in order to deal with the above mentioned problems. The Venus-program resulting from the replacement of free occurrences of `x` in E_1 with E_2, written:

$$E_1[\mathtt{x} := E_2]$$

is defined by cases on E_1.

$$
\begin{array}{rcl}
\mathtt{V}[x := E] & \equiv & \mathtt{V} \\
\mathtt{y}[x := E] & \equiv & \mathtt{y} \\
\mathtt{x}[x := E] & \equiv & E \\
(E_1 \ \mathtt{op}\ E_2)[x := E] & \equiv & (E_1[x := E]\ \mathtt{op}\ E_2[x := E]) \\
\mathtt{i2r}(E_1)[x := E_2] & \equiv & \mathtt{i2r}(E_1[x := E_2]) \\
\mathtt{r2i}(E_1)[x := E_2] & \equiv & \mathtt{r2i}(E_1[x := E_2]) \\
(\mathtt{let}\ \mathtt{x} = E_1\ \mathtt{in}\ E_2)[x := E_3] & \equiv & (\mathtt{let}\ \mathtt{x} = E_1[x := E_3]\ \mathtt{in}\ E_2) \\
(\mathtt{let}\ \mathtt{y} = E_1\ \mathtt{in}\ E_2)[x := E_3] & \equiv & (\mathtt{let}\ \mathtt{z} = E_1[x := E_3]\ \mathtt{in}\ E_2[y := z][x := E_3])\ \mathtt{z}\ \text{fresh}
\end{array}
$$

3 Call-by-Value and Call-by-Name Dynamic Semantics with Substitution

A given Venus-program will generally have a number of subexpressions available for simplification. For example, in

```
let x = 2 + 3 in let y = x + 4 in y * x
```

one could first simplify `2 + 3` to `5`, giving

```
let x = 5 in let y = x + 4 in y * x
```

Or one could plug the whole lot `2 + 3` in for the free occurrences of `x` in `let y = ...` giving

```
let y = (2 + 3) + 4 in y * (2 + 3)
```

The former strategy is called *call-by-value* evaluation, the latter is called *call-by-name* evaluation. Most, but not all, modern programming languages use call-by-value evaluation. Both strategies have advantages and disadvantages. Call-by-value is much simpler to implement, it makes it easier to reason about resource consumption and it's easier to reconcile with state and mutation. Call-by-value has the minor disadvantage that on rare occasions it will yield a spurious error.

Call-by-name evaluation, on the other hand, will always find an answer if there is one. But it's hard to implement efficiently and if implemented naively, call-by-name evaluation will do a great deal of unnecessary work.

Other simplification orders are possible. A variation of call-by-name reduces the inner `let`-form first. Or, one could use fewer steps by reducing subexpressions in parallel.

Whatever the chosen strategy, a dynamic evaluation system can be defined as a relation between Venus-programs and values:

$$\text{eval} \subseteq (E \times V)$$

$$\frac{}{\text{axiomatic judgement}}\;(\mathtt{axiom}) \qquad\qquad \frac{\text{... antecedents ...}}{\text{conclusion judgement}}\;(\ \mathtt{inference\ rule})$$

Figure 2: Axioms and Inference Rules

$$\frac{}{\vdash \mathtt{i} \Downarrow \mathtt{i}}\;(\mathtt{int}) \qquad\qquad \frac{}{\vdash \mathtt{r} \Downarrow \mathtt{r}}\;(\mathtt{real})$$

$$\frac{\vdash \mathtt{E} \Downarrow \mathtt{i};\ \mathtt{i2r(i)} = \mathtt{r}}{\vdash \mathtt{i2r(E)} \Downarrow \mathtt{r}}\;(\mathtt{i2r}) \qquad\qquad \frac{\vdash \mathtt{E} \Downarrow \mathtt{r};\ \mathtt{r2i(r)} = \mathtt{i}}{\vdash \mathtt{r2i(E)} \Downarrow \mathtt{i}}\;(\mathtt{r2i})$$

$$\frac{\vdash \mathtt{E}_1 \Downarrow \mathtt{V}_1;\ \vdash \mathtt{E}_2 \Downarrow \mathtt{V}_2;\ \mathrm{Apply}_2(\mathtt{o}, \mathtt{V}_1, \mathtt{V}_2) = \mathtt{V}}{\vdash \mathtt{E}_1\ \mathtt{o}\ \mathtt{E}_2 \Downarrow \mathtt{V}}\;(\mathtt{BinOp})$$

$$\frac{\vdash \mathtt{E}_1 \Downarrow \mathtt{V}';\ \vdash \mathtt{E}_2[x := V'] \Downarrow \mathtt{V}}{\vdash \mathtt{let\ x} : \tau = \mathtt{E}_1\ \mathtt{in}\ \mathtt{E}_2 \Downarrow \mathtt{V}}\;(\mathtt{CBV}) \qquad\qquad \frac{\vdash \mathtt{E}_2[x := E_1] \Downarrow \mathtt{V}}{\vdash \mathtt{let\ x} : \tau = \mathtt{E}_1\ \mathtt{in}\ \mathtt{E}_2 \Downarrow \mathtt{V}}\;(\mathtt{CBN})$$

Figure 3: $\vdash \mathtt{E} \Downarrow V$ Call-by-Value and Call-by-Name Evaluation of closed Venus-programs

Of course we naturally expect this relation to be a *function* from Venus-programs to values. The eval relation can be defined using an *axiomatic system* that derives *evaluation judgements* from a set of axioms and inference rules of the kind shown in Figure 2. These rules are most useful when they are expressed as step-by-step simplification rules, what is known in the trade as *small-step* or *structural* operational semantics (SOS).

It's somewhat easier to specify the evaluation relation by a so-called *big step* (aka *natural*) operational semantics which relates programs directly to their values without specifying the individual computation steps required in the process of getting from E to V. In big step semantics, we use judgements of the form $\vdash E \Downarrow V$.

$$\text{eval} = \{(E, V) \mid \vdash E \Downarrow V \text{ is derivable}\}$$

The dynamic semantics of Venus under both call-by-value evaluation and call-by-name evaluation is given in Figure 3. For call-by-value, choose the `CBV let` rule; for call-by-name choose the `CBN` rule for `let`.

4 Evaluation of Open Programs

The call-by-value and call-by-name semantics defined in Figure 3 both assign meanings to closed programs but not to open ones. In order to accommodate open programs we'll evaluate Venus-programs relative to a *value environment* A. Value environments are defined as follows:

$$A ::= \epsilon \mid A[\mathtt{x} \mapsto \mathtt{V}]$$

where the symbol ϵ denotes the empty environment and $A[\mathtt{x} \mapsto \mathtt{V}]$ denotes the extension of A that binds the symbol `x` to the value `V`. The notation, $A(\mathtt{x})$, denotes the value `V`, if it exists, that environment A assigns to identifier `x`. Values are looked up in an environment from right to left. More formally,

$$\begin{array}{rcl} \epsilon(\mathtt{x}) & = & \mathtt{unbound} \\ A[\mathtt{x} \mapsto \mathtt{V}](\mathtt{x}') & = & \begin{cases} \mathtt{V} & \text{if } \mathtt{x} = \mathtt{x}' \\ A(\mathtt{x}') & \text{otherwise.} \end{cases} \end{array}$$

We'll use $\mathrm{DomDef}(A)$ to denote the set of all variables bound in A.

The dynamic semantics of Venus-programs can be defined by an axiomatic system as before but with judgements that now include an environment A to the left of the turnstyle:

$$A \vdash \mathtt{E} \Downarrow \mathtt{V}$$

and

$$A \vdash D \Rightarrow A'$$

where $A \vdash D \Rightarrow A'$ means that processing the declaration D in environment A yields the extended environment A'.

A call-by-value semantics relating Venus-programs to their values is given as before by a *big step* semantics shown in Figures 4 and 5. This semantics is relatively straightforward.

If we want to specify a call-by-name semantics with environments, **as they did in the original Algol 60**, then we'll have to take steps to avoid name capture analogous to what we did with the renaming in the last clause of the definition of substitution. For example, in the program `let x = E1 in E2`, free occurrences of `x` in `E1`, would be captured if `E1` was evaluated in the environment constructed for `E2`. In order to avoid this problem, we'll introduce a **closure** structure of the form $\mathtt{Closure}(A, E)$ which is intended to allow for the correct evaluation of the free variables in E. In order to make this work, we extend our definition of values as follows:

```
V ::= i | r | Closure(A, E)
```

where `Closure(A, E)` is a structure combining an expression `E` with its definition environment `A`. We extend the rules as shown in Figures 4 and 7.

You would be wise to wonder about the connections between these various systems. Does the call-by-value version of the substitution semantics define the same evaluation relation as the call-by-value environment semantics? This is an interesting question that would be explored in a decent graduate class on programming languages.

$$\frac{}{A \vdash \texttt{i} \Downarrow \texttt{i}}\ (\texttt{int}) \qquad\qquad \frac{}{A \vdash \texttt{r} \Downarrow \texttt{r}}\ (\texttt{real})$$

$$\frac{A \vdash \texttt{E} \Downarrow \texttt{i};\ \ \texttt{i2r}(\texttt{i}) = \texttt{r}}{A \vdash \texttt{i2r}(\texttt{E}) \Downarrow \texttt{r}}\ (\texttt{i2r}) \qquad\qquad \frac{A \vdash \texttt{E} \Downarrow \texttt{r};\ \ \texttt{r2i}(\texttt{i}) = \texttt{i}}{A \vdash \texttt{r2i}(\texttt{E}) \Downarrow \texttt{i}}\ (\texttt{r2i})$$

$$\frac{A \vdash \texttt{E}_1 \Downarrow \texttt{V}_1;\ \ A \vdash \texttt{E}_2 \Downarrow \texttt{V}_2;\ \ \mathrm{Apply}_2(\texttt{o}, \texttt{V}_1, \texttt{V}_2) = \texttt{V}}{A \vdash \texttt{E}_1\ \texttt{o}\ \texttt{E}_2 \Downarrow \texttt{V}}\ (\texttt{BinOp})$$

$$\frac{A(\texttt{x}) = \texttt{V}}{A \vdash \texttt{x} \Downarrow \texttt{V}}\ (\texttt{Id})$$

$$\frac{A \vdash D \Rightarrow A';\ \ A' \vdash \texttt{E} \Downarrow \texttt{V}}{A \vdash \texttt{let}\ D\ \texttt{in}\ \texttt{E} \Downarrow \texttt{V}}\ (\texttt{Let})$$

Figure 4: $A \vdash \texttt{E} \Downarrow \texttt{V}$

$$\frac{A \vdash \texttt{E} \Downarrow \texttt{V}}{A \vdash \texttt{x} : \tau = \texttt{E} \Rightarrow A[\texttt{x} \mapsto \texttt{V}]}\ (\texttt{Decl})$$

Figure 5: $A \vdash \texttt{E} \Downarrow \texttt{V}$ where $\mathrm{DomDef}(A) \supseteq \mathrm{FV}(E)$, Call-by-Value.

$$\frac{}{A \vdash \mathtt{i} \Downarrow \mathtt{i}}\;(\mathtt{int}) \qquad \frac{}{A \vdash \mathtt{r} \Downarrow \mathtt{r}}\;(\mathtt{real})$$

$$\frac{A \vdash \mathtt{E} \Downarrow \mathtt{i};\ \mathtt{i2r}(\mathtt{i}) = \mathtt{r}}{A \vdash \mathtt{i2r}(\mathtt{E}) \Downarrow \mathtt{r}}\;(\mathtt{i2r}) \qquad \frac{A \vdash \mathtt{E} \Downarrow \mathtt{r};\ \mathtt{r2i}(\mathtt{i}) = \mathtt{i}}{A \vdash \mathtt{r2i}(\mathtt{E}) \Downarrow \mathtt{i}}\;(\mathtt{r2i})$$

$$\frac{A \vdash \mathtt{E}_1 \Downarrow \mathtt{V}_1;\ A \vdash \mathtt{E}_2 \Downarrow \mathtt{V}_2;\ \mathrm{Apply}_2(\mathtt{o}, \mathtt{V}_1, \mathtt{V}_2) = \mathtt{V}}{A \vdash \mathtt{E}_1\ \mathtt{o}\ \mathtt{E}_2 \Downarrow \mathtt{V}}\;(\mathtt{BinOp})$$

$$\frac{A(\mathtt{x}) = \mathtt{Closure}(A', \mathtt{E});\ A' \vdash \mathtt{E} \Downarrow \mathtt{V}}{A \vdash \mathtt{x} \Downarrow \mathtt{V}}\;(\mathtt{Id})$$

$$\frac{A \vdash D \Rightarrow A';\ A' \vdash \mathtt{E} \Downarrow \mathtt{V}}{A \vdash \mathtt{let}\ D\ \mathtt{in}\ \mathtt{E} \Downarrow \mathtt{V}}\;(\mathtt{Let})$$

Figure 6: $A \vdash \mathtt{E} \Downarrow \mathtt{V}$

$$\frac{}{A \vdash \mathtt{x} : \tau = \mathtt{E} \Rightarrow A[\mathtt{x} \mapsto \mathtt{Closure}(A, E)]}\;(\mathtt{Decl})$$

Figure 7: $A \vdash \mathtt{E} \Downarrow \mathtt{V}$ where $\mathrm{DomDef}(A) \supseteq \mathrm{FV}(E)$, Call-by-Name.

5 Type Discipline

Notwithstanding its simplicity, `Venus` is also a good vehicle for considering type structure. Type structure can play an important role in enhancing the reliability, the efficiency and the scalability of software. Types allow the programmer to think clearly about how to structure their code and to avoid errors. For example, the integer division operator is undefined when the divisor is zero. So the expression `(1 / 0)` is undefined and should cause an error. This is an example of a large class of errors that are generally impossible to detect without actually running the program. Division by zero is usually detected or *trapped* at run-time with a recoverable software fault. Other run-time errors such as *overflow* or *underflow* are detected by *hardware traps*.

Of course there are many other sorts of errors that might occur in our programs that are not trapped at run-time. For example, given a Java array

```
int a[] = new int[N]
```

and an array reference `a[exp]`, it is in general impossible for a compiler to determine whether or not the value of `exp` is a valid index into the array `a`. If this error is to be detected, it must be done at run-time.

Another class of unchecked errors, perhaps the most common kind of error, involves the programmer's mis-use of values. For example, misapplying the integer multiplication operator `*` to a boolean as in `True * 8` or attempting to apply an integer `12` as though it was a function `12(8)`, are both unchecked errors. Programs containing these kinds of "mismatch" (or square-peg in a round-hole) errors are said to be *ill-typed* or equivalently *not well-typed*.

A programming language that rules out unchecked errors is said to be *type safe* or sometimes *strongly typed*. A language can establish safety either by performing checks exclusively at run-time or by a combination of checks performed at compile-time and run-time The property of performing all such checks at run-time is called *dynamic typing* and languages such as Scheme that have this property are said to be *dynamically typed*. The property of ruling out as many mismatch errors as possible at compile-time and leaving over as few checks as possible for run-time is called *static typing* and languages such as Java, C#, Swift, Rust, Go and ML that have this property are said to be *statically typed*.

Overloading

Returning to the mis-application of operators, in many programming languages operators can be *overloaded*, that is, the same operator symbol can be used on operands of different types. In a dynamically typed languages operator overloading is resolved at run-time while in most statically typed languages, it is resolved at compile-time.

In order to resolve operator overloading at run-time, the run-time code must check the types of an operator's operands to determine which form of the operator to apply. E.g., given `2 + 3`, the language dynamically selects the integer form of addition while give `2.0 + 3` the language first dynamically coerces `3` to a float and then performs floating point addition. In order to determine which operator to apply, dynamically typed language require each value to have some run-time representation of their type. Usually this is in the form of *tag* bits that specify which summand of a sum type the value represents. It should be noted that the management and checking of tag bits requires extra time and space.

In a statically typed language such as Java, the `+` operator performs addition on various sorts of numbers and it performs string concatenation. When a Java programmer types `a + b` when `a` and `b` are both of type `int`, integer addition will take place. When the programmer types `a + b` when `a` and `b` are both of type `float`, floating point addition will occur. And when the operands are of mixed type, the Java compiler inserts type coercions that promote the value of the less informative type (i.e., `int`) to the more informative type (i.e., `float`). It is important to note that the required type conversions are determined at compile-time so there is generally no requirement for run-time tag bits.

The language Venus does not support operator overloading. Venusfeatures integer operators `{+, -, *, /, %}` and separate real operators `{+., -., *., /.}`. So the following is ill-typed:

```
let x = 2 +. 3.0
in
x + x
```

and Venus does not automatically insert type coercions for the programmer. Venus does feature two explicit type conversion operators `i2r` (i.e., integer to real) and `r2i` (i.e., real to integer). So the following version containing explicit coercions is well-typed:

```
let x = i2r(2) +. 3.0
in
r2i(x) + r2i(x)
```

5.1 Strongly Typed Venus

As we have discussed previously, in a **strongly typed** programming language the implementation checks expressions for consistent usage in order to ensure that *untrapped errors* do not occur at run-time. There are two major approaches to strong typing: **purely dynamic checking** and **mostly static checking**.

Dynamically-Typed Venus

In this setting, a Venus compiler generates code in such a way that run-time values are annotated with extra bits encoding type information and this information is checked during program execution to ensure consistency. For example, in the implementation of `v1 + v2`, wrapper code around the `+` operation would check to ensure that both `v1` and `v2` were of the correct type, if not a trap would occur. If they were of the correct type, the addition would take place and the resulting sum would be annotated with the appropriate type information.

This incurs run-time over head and delays the detection of errors until run-time. But it is safe in the sense that the code can avoid untrapped errors.

5.2 Statically-Typed Venus

In the statically-typed setting, symbols in the source language are associated with type annotations and the consistency of program phrases are inferred at compile-time from these annotations. The only form of run-time checking is to trap errors such as *division by zero* or *array bounds* that cannot be detected at compile-time because they are undecidable.

There are two ways that type information can be related to program symbols:

- **explicitly, ala Church**: the programmer provides explicit type annotations that can be checked by the compiler during compilation;
- **implicitly, ala Curry**: the programmer doesn't always provide explicit type annotations but the type annotations for symbols can be *inferred* at compile-time by the usage of the symbols in the code.

5.3 Typing Checking with Explicit Types

In order to facilitate static type checking for Venus, we're going to banish the declaration class that omits the type. All binding occurrences of variables are required to have explicit types as in

```
let y : int = let x : float = i2r(2) +. 3.0
              in
              r2i(x) + r2i(x)
in
y
```

5.3.1 Type Environments

The static type system of Venus-programs will be defined as usual by an axiomatic system. This is often referred to as the **static semantics**. Judgements will include a *type environment* Γ that assigns types to the free variables in the program and additionally to the primitive operations in Venus. A type environment is a finite function from Operator to τ. The structure of a type environment Γ is defined as follows:

$$\Gamma ::= \epsilon \mid \Gamma[\mathtt{x} : \tau]$$

Type environments are defined by analogy with value environments so we won't repeat the details. In order to simplify type checking of operators, we're going to extend the abstract syntax of types τ as follows:

$$\tau ::= \mathtt{int} \mid \mathtt{real} \mid \tau \to \tau \mid \tau{\times}\tau$$

Although Venus doesn't have any values of product type or function type, the types $\tau \times \tau$ and $\tau \to \tau$ can be used to represent the types for binary primitive operations. We can

$$\frac{}{\Gamma \vdash \texttt{i} : \texttt{int}}\,(\texttt{int}) \qquad\qquad \frac{}{\Gamma \vdash \texttt{r} : \texttt{real}}\,(\texttt{real})$$

$$\frac{\Gamma \vdash \texttt{E} : \texttt{int}}{\Gamma \vdash \texttt{i2r(E)} : \texttt{real}}\,(\texttt{i2r}) \qquad\qquad \frac{\Gamma \vdash \texttt{E} : \texttt{real}}{\Gamma \vdash \texttt{r2i(E)} : \texttt{int}}\,(\texttt{r2i})$$

$$\frac{\Gamma(\texttt{o}) = \tau_1 \times \tau_2 \to \tau;\ \Gamma \vdash \texttt{E}_1 : \tau_1;\ \Gamma \vdash \texttt{E}_2 : \tau_2}{\Gamma \vdash \texttt{E}_1\ \texttt{o}\ \texttt{E}_2 : \tau}\,(\texttt{BinOp})$$

$$\frac{\Gamma(\texttt{x}) = \tau}{\Gamma \vdash \texttt{x} : \tau}\,(\texttt{Id})$$

$$\frac{\Gamma \vdash D \Rightarrow \Gamma';\ \Gamma' \vdash \texttt{E} : \tau}{\Gamma \vdash \texttt{let}\ D\ \texttt{in}\ \texttt{E} : \tau}\,(\texttt{Let})$$

Figure 8: $\Gamma \vdash \texttt{E} : \tau$.

start with an initial environment that has types built-in for these built-in operators. In the remainder of this document, Γ_0 will stand for the particular type environment:

$$\Gamma_0 = \epsilon[+ : \texttt{int} \times \texttt{int} \to \texttt{int}][* : \texttt{int} \times \texttt{int} \to \texttt{int}] \cdots\cdots$$

5.3.2 Static Semantics

The well-typedness relation is a system for deriving *type judgements* of the form:

$$\Gamma \vdash \texttt{E} : \tau$$

and

$$\Gamma \vdash \texttt{D} \Rightarrow \Gamma$$

The axioms and inference rules defining the relation are defined in figures 8 and 9. Intuitively, if a type judgment $\Gamma \vdash \texttt{E} : \tau$ is derivable using the axioms and inference rules then the triple $(\Gamma, \texttt{E}, \tau)$ is in the well-typedness relation.

Of course, we would want it to be the case that if the type system concludes that Venus-program E is of type τ, then if we run E through any of the dynamic semantics it would produce a value of the τ. This important property is called **type soundness**.

5.4 Type Inference

Whether or not the coder has provided explicit type assertions in their code, some amount of type inference is required in a compiler for a statically typed programming language.

$$\frac{\Gamma \vdash \mathtt{E} : \tau}{\Gamma \vdash \mathtt{x} : \tau = \mathtt{E} \Rightarrow \Gamma[x : \tau]} \ (\mathtt{Decl})$$

Figure 9: $\Gamma \vdash D \Rightarrow \Gamma'$.

For example, in `let x : int = 2 in x * 10`, the compiler has to infer that the composite expression `(x * 10)` is of type `int`.

The type inference algorithm that we'll use is due to Mitch Wand. Before we define the algorithm, we'll cover some material on *unification*.

5.4.1 Unification

Let α_1, α_2, etc be a set of *type variables* and define the set of *type expressions* as the extension of types that include type variables.

$$\tau ::= \alpha \mid \mathtt{int} \mid \mathtt{real} \mid \tau \rightarrow \tau \mid \tau \times \tau$$

A *substitution* σ, θ, λ is a set of pairs τ/α, pronounced "τ for α" where α is not in τ

$$\sigma = \{\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n\}$$

We use ϵ for the empty substitution. The *application* of substitution σ to type expression τ, written $\tau\sigma$, is obtained by simultaneously replacing each α_i in τ by τ_i. For example, let

$$\sigma = \{\mathtt{int}/\alpha_1, \alpha_2 \rightarrow \mathtt{real}/\alpha_3\}$$

and let $\tau = \alpha_3 \rightarrow \alpha_1$, then $\tau\sigma = (\alpha_2 \rightarrow \mathtt{real}) \rightarrow \mathtt{int}$.

Let $\theta = \{\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n\}$, $\sigma = \{\tau_1'/\alpha_1', \ldots, \tau_m'/\alpha_m'\}$. The *composition* of θ and σ, written $\theta \circ \sigma$ is obtained from the set

$$\{\tau_1\sigma/\alpha_1, \ldots, \tau_n\sigma/\alpha_n, \tau_1'/\alpha_1', \ldots, \tau_m'/\alpha_m'\}$$

by deleting any pair $\tau_j\sigma/\alpha_j$ for which $\tau_j\sigma = \alpha_j$ and any pair τ_i'/α_i' such that $\alpha_i' \in \{\alpha_1, \ldots, \alpha_n\}$.

A substitution θ is a *unifier* of a set of type expressions $\{\tau_1, \ldots, \tau_n\}$ if and only if $\tau_1\theta = \ldots = \tau_n\theta$. A substitution σ is a *most general unifier* if and only if for any unifier θ, there exists a substitution λ such that $\theta = \sigma \circ \lambda$.

Disagreement

Let $W = \{\tau_1, \ldots, \tau_n\}$. The *disagreement set* of W is the set of type expressions $D = \{\tau_1', \ldots, \tau_n'\}$ such that τ_i' is the largest subtree of τ_i such that $\tau_i' \neq \ldots \neq \tau_n'$. For example, if

$$W = \{\mathtt{int} \rightarrow \mathtt{int} \times \alpha_1, \mathtt{int} \rightarrow \alpha_2, \mathtt{int} \rightarrow \alpha_3 \times \mathtt{real}\}$$

Then $D = \{\mathtt{int} \times \alpha_1, \alpha_2, \alpha_3 \times \mathtt{real}\}$.

The Unification Algorithm

The input is a set of type expressions $W = \{\tau_1, \ldots, \tau_n\}$.

1. Set $k = 0$, $W_0 = W$ and $\sigma_0 = \epsilon$.
2. If W_k is a singleton, stop; σ_k is a most general unifier of W. Otherwise let D_k be the disagreement set of W_k.
3. If D_k does not have elements α and τ such that α does not occur in τ, stop; W is not unifiable.
4. Otherwise, let $\sigma_{k+1} = \sigma_k \circ \{\tau/\alpha\}$ and $W_{k+1} = W_k\{\tau/\alpha\}$. Set $k = k + 1$ and go to step 2.

5.4.2 Wand's Algorithm

Wand's algorithm generates a set of equations $E = \{\tau_1 = \tau_1', \ldots, \tau_n = \tau_n'\}$ between type expressions. Let Γ be a map from variables to type expressions. Let $\texttt{E}_0$ be a Venus-program, let Γ_0 be a type expression map that maps the built-in operators to their types and all free varaibles in $\texttt{E}_0$ to fresh type variables. Let α be a fresh type variable.

1. Let $E = \{\}$ and let $G_0 = \{(\Gamma_0, \texttt{E}_0, \alpha)\}$.
2. While $G \neq \{\}$, remove a subgoal $(\Gamma, \texttt{E}, \tau)$ from G and add equations to E and subgoals to G by cases on $\texttt{E}$:
    - $\texttt{E} = \texttt{i}$: Add $\tau = \texttt{int}$ to E.
    - $\texttt{E} = \texttt{r}$: Add $\tau = \texttt{real}$ to E.
    - $\texttt{E} = \texttt{x}$: Add $\tau = \Gamma(\texttt{x})$ to E.
    - $\texttt{E} = \texttt{i2r}(\texttt{E}')$: Add $\tau = \texttt{real}$ to E and $(\Gamma, \texttt{E}', \texttt{int})$ to G.
    - $\texttt{E} = \texttt{r2i}(\texttt{E}')$: Add $\tau = \texttt{int}$ to E and $(\Gamma, \texttt{E}', \texttt{real})$ to G.
    - $\texttt{E} = (\texttt{E}_1 \ \texttt{o} \ \texttt{E}_2)$: where $\Gamma(\texttt{o}) = \tau_1 \times \tau_2 \to \tau_3$. Add $\tau = \tau_3$ to E and $(\Gamma, \texttt{E}_1, \tau_1)$ and $(\Gamma, \texttt{E}_2, \tau_2)$ to G.
    - $\texttt{E} = \texttt{let x} = \texttt{E}_1 \ \texttt{in} \ \texttt{E}_2$: Let α_1 and α_2 be fresh type variables. Add $\tau = \alpha_2$ to E and $(\Gamma[\texttt{x} : \alpha_1], \texttt{E}_2, \alpha_2)$ and $(\Gamma, \texttt{E}_1, \alpha_1)$ to G.
3. $G = \{\}$, return E.

5.4.3 Piecing it Together

For Venus-program $\texttt{E}$ run Wand's algorithm to produce $E = \{\tau_1 = \tau_1', \tau_2 = \tau_2', \ldots, \tau_n = \tau_n'\}$. Let $\sigma_0 = \epsilon$ and for each $i \in \{1, \ldots n\}$, let $\sigma_i = \text{Unify}(\{\tau_i, \tau_i'\}, \sigma_{i-1})$. If this process produces σ_n, this is the most general unifier for E.