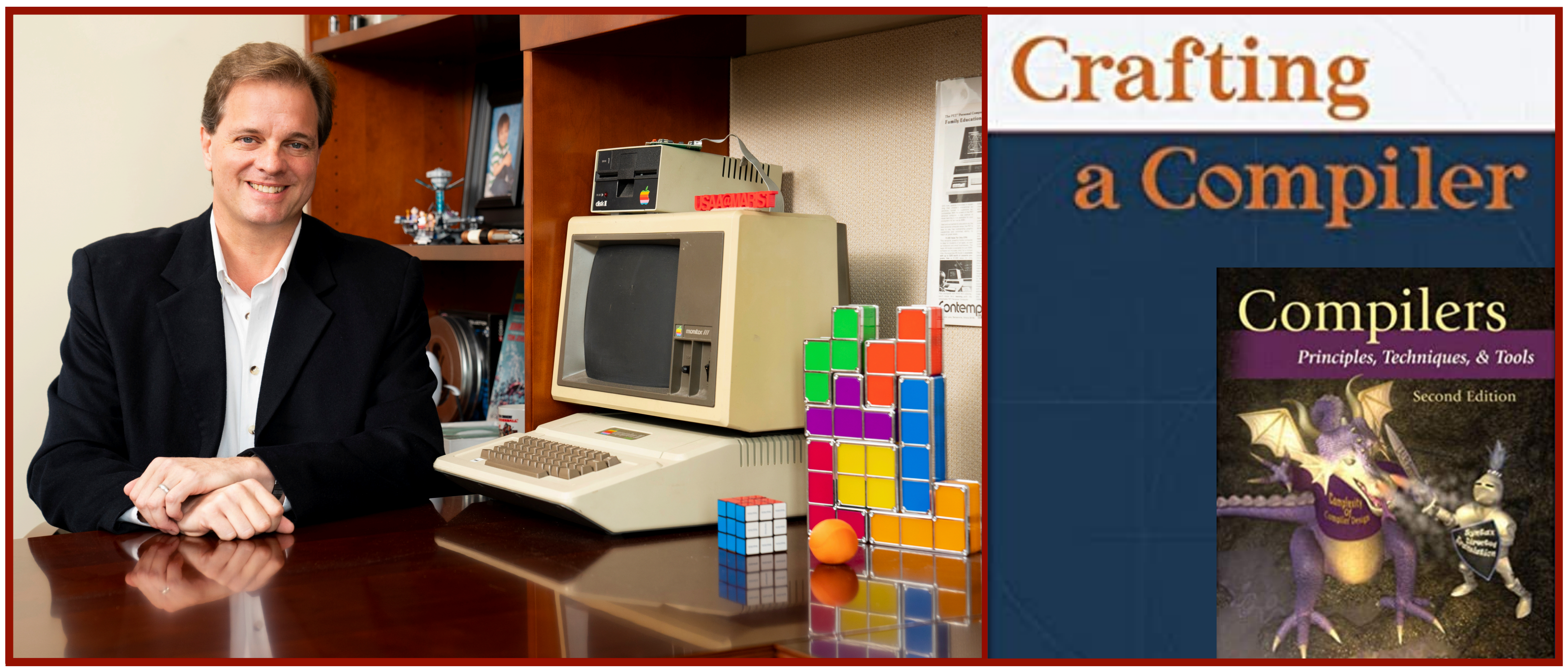
Type Systems



Alan G. Labouseur, Ph.D.
Alan.Labouseur@Marist.edu

Type Systems

The Basics

What is a Type?

- A set of values
- A set of operations on those values

Type errors happen when we try to perform operations on values that do not support them.

Type expressions are textual representations of type

- primitive: *int*, *boolean*, *real*, *date*, *time*, *char*, *pointer*, …
- complex: *timestamp*, *latitude*, *longitude*, *student-id*, …

  What about *string*? Or *String*?

Type Systems

The Basics

What is a Type?

- A set of values
- A set of operations on those values

Type errors happen when we try to perform operations on values that do not support them.

Type expressions are textual representations of type

- primitive: *int*, *boolean*, *real*, *date*, *time*, *char*, *pointer*, …
- complex: *string*, *timestamp*, *latitude*, *longitude*, *student-id*, …

Type systems consist of rules governing what operations are permitted on what values.

- strong type systems prevent type errors at runtime.
- weak type systems ~~allow~~ encourage type errors at runtime.
- Type systems can be documented and reasoned about using inferences rules.

Type Systems

Specifying a Type System with Inference Rules

$$\frac{\textit{preconditions}}{\textit{postconditions}}$$

We can use inference rules from mathematics and Axiomatic Semantics. Why?

- It's fun.
- It's accurate. We need a rigorous definition of types and type systems so that we can enforce them in the compiler.
- It gives flexibility in implementation because it's not tied to any grammar.
- It allows for formal verification of program properties.
- It's what used in the computer science literature.

Inference Rules

An inference rule is written

$$\frac{f_1,\ f_2,\ \ldots f_n}{f_0}$$

It expresses that **if** f_1, f_2, … f_n are theorems — that is, they are proven well-formed formulae (WFF) — **then** we can infer that f_0 is another theorem.

That's nice, but how do we know?
How can we actually prove things?

Let's look at famous inference rule: Modus Ponens.

A Famous Inference Rule

Modus Ponens

$$\frac{p,\ p \Rightarrow q}{q}$$

Modus Ponens ("the mode that affirms") can be read:

if

we have p (meaning, p is true) **and**

p implies q

then

we can infer that q is true.

end if

The implication/conditional operator ($\Rightarrow$) is like a contract:
if p then q.

Inference Rule vs. Propositional Connective

Modus Ponens

$$\frac{p,\ p \Rightarrow q}{q}$$

Inference Rule

Modus Ponens ("the mode that affirms") can be read:

if

we have p (meaning, p is true) **and**

p implies q

then

we can infer that q is true.

end if

Propositional Connective

The implication/conditional operator ($\Rightarrow$) is like a contract:
if p then q.

Let's review this in Propositional logic.

Propositional Logic

Truth Tables

p	q
0	0
0	1
1	0
1	1

Propositional logic has only false and true, no variables.
It also has logical operators like and, or, and not.
(propositional connectors)

Propositional Logic

Truth Tables

p	q	p∧q
0	0	0
0	1	0
1	0	0
1	1	1

Propositional logic has only false and true, no variables.
It also has logical operators like **and**, or, and not.

Propositional Logic

Truth Tables

p	q	p∧q	p∨q
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Propositional logic has only false and true, no variables.
It also has logical operators like and, **or**, and not.

Propositional Logic

Truth Tables

p	q	p∧q	p∨q	¬p
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Propositional logic has only false and true, no variables.
It also has logical operators like and, or, and **not**.

Do we need more? (Do we even need all of these?)

Propositional Logic

Truth Tables

p	q	$p \wedge q$	$p \vee q$	$\neg p$
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Propositional logic has only false and true, no variables.
It also has logical operators like and, or, and not.

Do we need more? No. (Do we even need all of these? No.)

$$p \vee q = \neg(\neg p \wedge \neg q)$$

Propositional Logic

Truth Tables

p	q	p∧q	p∨q	¬p	p⇒q
0	0	0	0	1	
0	1	0	1	1	
1	0	0	1	0	
1	1	1	1	0	

Propositional logic has only false and true, no variables.
It also has logical operators like and, or, and not.

Implication is like a contract:
"if p then q" or "$p \Rightarrow q$".

Propositional Logic

Truth Tables

p	q	p∧q	p∨q	¬p	p⇒q
0	0	0	0	1	1
0	1	0	1	1	1
1	0	0	1	0	
1	1	1	1	0	

These are vacuously true because *p* is false and false can imply anything because it's an invalid premise.

Also, we take "if *p* then *q*" to be false **only** when *p* is true and *q* is false.

Propositional logic has only false and true, no variables.
It also has logical operators like and, or, and not.

Implication is like a contract:
"if *p* then *q*" or "$p \Rightarrow q$".

Propositional Logic

Truth Tables

p	q	p∧q	p∨q	¬p	p⇒q
0	0	0	0	1	1
0	1	0	1	1	1
1	0	0	1	0	0
1	1	1	1	0	

This is false because *p* is true and *q* is false, and "true implies false" is false.

Propositional logic has only false and true, no variables.
It also has logical operators like and, or, and not.

Implication is like a contract:
"if *p* then *q*" or "$p \Rightarrow q$".

Propositional Logic

Truth Tables

p	q	p∧q	p∨q	¬p	p⇒q
0	0	0	0	1	1
0	1	0	1	1	1
1	0	0	1	0	0
1	1	1	1	0	1

This is true because *p* is true and *q* is true, and “true implies true” is true.

Propositional logic has only false and true, no variables.
It also has logical operators like and, or, and not.

Implication is like a contract:
“if *p* then *q*” or “$p \Rightarrow q$”.

Propositional Logic

Truth Tables

p	q	p∧q	p∨q	¬p	p⇒q	¬p∨q
0	0	0	0	1	1	
0	1	0	1	1	1	
1	0	0	1	0	0	
1	1	1	1	0	1	

Propositional logic has only false and true, no variables.
It also has logical operators like and, or, and not.

Implication is like a contract:
"if p then q" or "$p \Rightarrow q$".
Implication can also be written as $\neg p \vee q$.

Propositional Logic

Truth Tables

These two columns are the same.
Both are implication.

p	q	p∧q	p∨q	¬p	p⇒q	¬p∨q
0	0	0	0	1	1	1
0	1	0	1	1	1	1
1	0	0	1	0	0	0
1	1	1	1	0	1	1

Propositional logic has only false and true, no variables.
It also has logical operators like and, or, and not.

Implication is like a contract:
"if p then q" or "$p \Rightarrow q$".
Implication can also be written as $\neg p \vee q$.

Propositional Logic

Truth Tables

p	q	$p \wedge q$	$p \vee q$	$\neg p$	$p \Rightarrow q$	$\neg p \vee q$	$p \wedge q \Rightarrow (p \Rightarrow q)$
0	0	0	0	1	1	1	
0	1	0	1	1	1	1	
1	0	0	1	0	0	0	
1	1	1	1	0	1	1	

Propositional logic has only false and true, no variables.
It also has logical operators like and, or, and not.

Implication is like a contract:
"if p then q" or "$p \Rightarrow q$".
Implication can also be written as $\neg p \vee q$.

A tautology is logical statement that is always true regardless of the truth values of its components. Here's one: $p \wedge q \Rightarrow (p \Rightarrow q)$

Propositional Logic

Truth Tables

p	q	p∧q	p∨q	¬p	p⇒q	¬p∨q	p∧q⇒(p⇒q)
0	0	0	0	1	1	1	1
0	1	0	1	1	1	1	1
1	0	0	1	0	0	0	1
1	1	1	1	0	1	1	1

Tautology

Propositional logic has only false and true, no variables.
It also has logical operators like and, or, and not.

Implication is like a contract:
"if p then q" or "$p \Rightarrow q$".
Implication can also be written as $\neg p \vee q$.

A tautology is logical statement that is always true regardless of the truth values of its components. Here's one: $p \wedge q \Rightarrow (p \Rightarrow q)$

Propositional Logic

Truth Tables

p	q	p∧q	p∨q	¬p	p⇒q	¬p∨q	p∧q⇒(p⇒q)
0	0	0	0	1	1	1	1
0	1	0	1	1	1	1	1
1	0	0	1	0	0	0	1
1	1	1	1	0	1	1	1

What's the opposite of a tautology, where the statement is always false?

Propositional logic has only false and true, no variables.
It also has logical operators like and, or, and not.

Implication is like a contract:
"if p then q" or "$p \Rightarrow q$".
Implication can also be written as $\neg p \vee q$.

A tautology is logical statement that is always true regardless of the truth values of its components. Here's one: $p \wedge q \Rightarrow (p \Rightarrow q)$

Propositional Logic

Truth Tables

p	q	p∧q	p∨q	¬p	p⇒q	¬p∨q	p∧q⇒(p⇒q)
0	0	0	0	1	1	1	1
0	1	0	1	1	1	1	1
1	0	0	1	0	0	0	1
1	1	1	1	0	1	1	1

What's the opposite of a tautology, where the statement is always false?
A contradiction.

Propositional logic has only false and true, no variables.
It also has logical operators like and, or, and not.

Implication is like a contract:
"if p then q" or "$p \Rightarrow q$".
Implication can also be written as $\neg p \vee q$.

A tautology is logical statement that is always true regardless of the truth values of its components. Here's one: $p \wedge q \Rightarrow (p \Rightarrow q)$

Propositional Logic

Truth Tables

A contradiction

p	q	p∧q	p∨q	¬p	p⇒q	¬p∨q	p∧q⇒(p⇒q)	p∧¬p
0	0	0	0	1	1	1	1	0
0	1	0	1	1	1	1	1	0
1	0	0	1	0	0	0	1	0
1	1	1	1	0	1	1	1	0

Contradictions cannot exist.

Propositional logic has only false and true, no
It also has logical operators like and, or, and n

Implication is like a contract:
"if p then q" or "$p \Rightarrow q$".
Implication can also be written as $\neg p \vee q$.

A tautology is logical statement that is always true regardless of the truth values of its components. Here's one: $p \wedge q \Rightarrow (p \Rightarrow q)$

Back to that Famous Inference Rule

Propositional Logic for Modus Ponens

p	q	p∧q	p∨q	¬p	p⇒q	¬p∨q	p∧q⇒(p⇒q)
0	0	0	0	1	1	1	1
0	1	0	1	1	1	1	1
1	0	0	1	0	0	0	1
1	1	1	1	0	1	1	1

Modus Ponens

$$\frac{p,\ p \Rightarrow q}{q}$$

can be written "if p and $p \Rightarrow q$ then q", which can be written

$$(\,p \ \wedge \ (p \Rightarrow q)\,) \ \Rightarrow q$$

A Famous Inference Rule

Propositional Logic for Modus Ponens

p	q	$p \wedge q$	$p \vee q$	$\neg p$	$p \Rightarrow q$	$\neg p \vee q$	$p \wedge q \Rightarrow (p \Rightarrow q)$	$p \wedge (p \Rightarrow q)$
0	0	0	0	1	1	1	1	0
0	1	0	1	1	1	1	1	0
1	0	0	1	0	0	0	1	0
1	1	1	1	0	1	1	1	1

Modus Ponens

$$\frac{p,\ p \Rightarrow q}{q}$$

can be written "if p and $p \Rightarrow q$ then q", which can be written

$$(\underline{p \wedge (p \Rightarrow q)}) \Rightarrow q$$

A Famous Inference Rule

Propositional Logic for Modus Ponens

p	q	p∧q	p∨q	¬p	p⇒q	¬p∨q	p∧q⇒(p⇒q)	p∧(p⇒q)	(p∧(p⇒q))⇒q
0	0	0	0	1	1	1	1	0	1
0	1	0	1	1	1	1	1	0	1
1	0	0	1	0	0	0	1	0	1
1	1	1	1	0	1	1	1	1	1

Modus Ponens

$$\frac{p,\ p \Rightarrow q}{q}$$

can be written "if p and $p \Rightarrow q$ then q", which can be written

$$(p \wedge (p \Rightarrow q)) \Rightarrow q$$

A Famous Inference Rule

Propositional Logic for Modus Ponens

p	q	p∧q	p∨q	¬p	p⇒q	¬p∨q	p∧q⇒(p⇒q)	p∧(p⇒q)	(p∧(p⇒q))⇒q
0	0	0	0	1	1	1	1	0	1
0	1	0	1	1	1	1	1	0	1
1	0	0	1	0	0	0	1	0	1
1	1	1	1	0	1	1	1	1	1

Modus Ponens

$$\frac{p,\ p \Rightarrow q}{q}$$

can be written "if p and $p \Rightarrow q$ then q", which can be written

$$(p \wedge (p \Rightarrow q)) \Rightarrow q$$

Tautology. Woot!

Inference Rules for Type Systems

With Modus Ponens proved and used as the basis for inference rules, we need to move from Propositional logic to Predicate logic.

The complexity of reasoning about type systems cannot be handled with truth tables because we need to accommodate ideas like *any*, *all*, or *some*. Also, we need variables and functions. This leads us to . . .

First Order Logic

- variables
- domains
- named constants
- relations (>, <, etc.)
- functions (math operations)
- logical operators
- quantifiers (for-all “∀” and there-exists “∃”)

Now we can reason about type systems.

Type Systems

Primitives / Literals / Intrinsic Types

Boolean literals

$$\frac{}{\vdash \mathit{true}\text{: boolean}} \qquad \frac{}{\vdash \mathit{false}\text{: boolean}}$$

An empty pre-condition means "under any circumstances".

String literals

$$\frac{s \text{ is a string literal or constant}}{\vdash s\text{: string}}$$

Integer literals

$$\frac{i \text{ is an integer literal or constant}}{\vdash i\text{: integer}}$$

Type Systems

Addition

Boolean literals

We cannot add Booleans because no inference rules are given to support that.

String literals

$$\frac{\vdash e_1 : \text{string} \qquad \vdash e_2 : \text{string}}{\vdash e_1 + e_2 : \text{string}}$$

This would be better labeled as *concatenation*, since it's not really addition. Maybe we should choose a different operator, like " • ".

Integer literals

$$\frac{\vdash e_1 : \text{integer} \qquad \vdash e_2 : \text{integer}}{\vdash e_1 + e_2 : \text{integer}}$$

Something is missing here . . .

Type Systems

Assignment

$$\frac{\vdash e_1 : \mathrm{T} \quad \vdash e_2 : \mathrm{T} \quad \mathrm{T}\text{ is a primitive type}}{\vdash e_1 = e_2 : \mathrm{T}}$$

Comparisons

$$\frac{\vdash e_1 : \mathrm{T} \quad \vdash e_2 : \mathrm{T} \quad \mathrm{T}\text{ is a primitive type}}{\vdash e_1 == e_2 : \text{boolean}}$$

$$\frac{\vdash e_1 : \mathrm{T} \quad \vdash e_2 : \mathrm{T} \quad \mathrm{T}\text{ is a primitive type}}{\vdash e_1 > e_2 : \text{boolean}}$$

$$\frac{\vdash e_1 : \mathrm{T} \quad \vdash e_2 : \mathrm{T} \quad \mathrm{T}\text{ is a primitive type}}{\vdash e_1 \mathrel{!=} e_2 : \text{boolean}}$$

$$\frac{\vdash e_1 : \mathrm{T} \quad \vdash e_2 : \mathrm{T} \quad \mathrm{T}\text{ is a primitive type}}{\vdash e_1 < e_2 : \text{boolean}}$$

I've got a bad feeling about this . . .

Type Systems

Example

```
string x = “I have a”;
string y = “bad feeling”;

int aboutThis(int x) {
   return x + y;
}

main() {
   int z;
   z = aboutThis(42);
   print(z);
}
```

Type Systems

Example

```
string x = "I have a";
string y = "bad feeling";

int aboutThis(int x) {
   return x + y;
}

main() {
   int z;
   z = aboutThis(42);
   print(z);
}
```

Things we know
x: string
y: string
x: int
z: int

Type Systems

Example

```
string x = "I have a";
string y = "bad feeling";

int aboutThis(int x) {
   return x + y;
}

main() {
   int z;
   z = aboutThis(42);
   print(z);
}
```

Things we know
x: string
y: string
x: int
z: int

Things we wonder about
Is x + y a legal operation under our type rules?

$$\frac{\vdash e_1 : \text{string} \qquad \vdash e_2 : \text{string}}{\vdash e_1 + e_2 : \text{string}} \qquad\qquad \frac{\vdash e_1 : \text{integer} \qquad \vdash e_2 : \text{integer}}{\vdash e_1 + e_2 : \text{integer}}$$

Type Systems

Example

```
string x = “I have a”;
string y = “bad feeling”;

int aboutThis(int x) {
   return x + y;
}

main() {
   int z;
   z = aboutThis(42);
   print(z);
}
```

Things we know
x: string
y: string
x: int
z: int

Things we wonder about
Is x + y a legal operation under our type rules?
Which x is that?

$$\frac{\vdash e_1 : \text{string} \qquad \vdash e_2 : \text{string}}{\vdash e_1 + e_2 : \text{string}} \qquad \frac{\vdash e_1 : \text{integer} \qquad \vdash e_2 : \text{integer}}{\vdash e_1 + e_2 : \text{integer}}$$

Type Systems

Example - No Context

```
string x = "I have a";
string y = "bad feeling";

int aboutThis(int x) {
   return x + y;
}

main() {
   int z;
   z = aboutThis(42);
   print(z);
}
```

Things we know
x: string
y: string
x: int
z: int

Things we wonder about
Is x + y a legal operation under our type rules?

$$\frac{\vdash e_1 : \text{string} \quad \vdash e_2 : \text{string}}{\vdash e_1 + e_2 : \text{string}} \qquad \frac{\vdash e_1 : \text{integer} \quad \vdash e_2 : \text{integer}}{\vdash e_1 + e_2 : \text{integer}}$$

The problem is that our type rules lack context. We need to strengthen them to specify under what circumstances they apply. In other words, we need **scope**.

Type Systems

Addition with Scope Context

Boolean literals

We cannot add Booleans because no inference rules are given to support that.

String literals

$$\frac{S \vdash e_1 : \text{string} \qquad S \vdash e_2 : \text{string}}{S \vdash e_1 \cdot e_2 : \text{string}}$$

e_1 is a string in scope S
e_2 is a string in scope S
$e_1 \cdot e_2$ results in a string in scope S

Integer literals

$$\frac{S \vdash e_1 : \text{integer} \qquad S \vdash e_2 : \text{integer}}{S \vdash e_1 + e_2 : \text{integer}}$$

e_1 is an integer in scope S
e_2 is an integer in scope S
$e_1 + e_2$ results in an integer in scope S

This is better . . .

Type Systems

Assignment
with Scope Context

$$\frac{S \vdash e_1 : T \quad S \vdash e_2 : T \quad T \text{ is a primitive type}}{S \vdash e_1 = e_2 : T}$$

Comparisons
with Scope Context

$$\frac{S \vdash e_1 : T \quad S \vdash e_2 : T \quad T \text{ is a primitive type}}{S \vdash e_1 == e_2 : \text{boolean}}$$

$$\frac{S \vdash e_1 : T \quad S \vdash e_2 : T \quad T \text{ is a primitive type}}{S \vdash e_1 > e_2 : \text{boolean}}$$

$$\frac{S \vdash e_1 : T \quad S \vdash e_2 : T \quad T \text{ is a primitive type}}{S \vdash e_1 \mathrel{!}= e_2 : \text{boolean}}$$

$$\frac{S \vdash e_1 : T \quad S \vdash e_2 : T \quad T \text{ is a primitive type}}{S \vdash e_1 < e_2 : \text{boolean}}$$

I’ve got a good feeling about this.

Type Systems

Addition with Scope Context

$$\frac{S \vdash e_1 : T \quad S \vdash e_2 : T \quad T \text{ is a primitive type}}{S \vdash e_1 + e_2 : T}$$

Implementation in Prolog

```
/* Symbol Table facts */
type(i, int).
type(j, int).
type(x, real).
type(y, real).

/* Type System Rules */
expectedtype(plus(E1,E2),T) :- type(E1,T),
                               type(E2,T).
/* Queries to infer/check type  */

expectedtype(plus(i,j),X)   /* int */

expectedtype(plus(x,y),X)   /* real */

expectedtype(plus(i,y),X)   /* false - Type error (No unifying match.) */
```

Type Systems

Addition
with Scope Context

$$\frac{S \vdash e_1 : T \qquad S \vdash e_2 : T \qquad T \text{ is a primitive type}}{S \vdash e_1 + e_2 : T}$$

Implementation in Prolog

SWISH File Edit Examples Help

Program

```prolog
/* Facts */
type(i, int).
type(j, int).
type(x, real).
type(y, real).

/* Rules */
expectedtype(plus(E1,E2),T) :- type(E1,T),
                               type(E2,T).
```

type(i, X)

X = int

```prolog
?- type(i, X)
```

Type Systems

Addition with Scope Context

$$\frac{\begin{array}{l} S \vdash e_1 : T \\ S \vdash e_2 : T \\ T \text{ is a primitive type} \end{array}}{S \vdash e_1 + e_2 : T}$$

Implementation in Prolog

SWISH File Edit Examples Help

Program

```prolog
/* Facts */
type(i, int).
type(j, int).
type(x, real).
type(y, real).

/* Rules */
expectedtype(plus(E1,E2),T) :- type(E1,T),
                               type(E2,T).
```

expectedtype(plus(i,j),X)

X = int

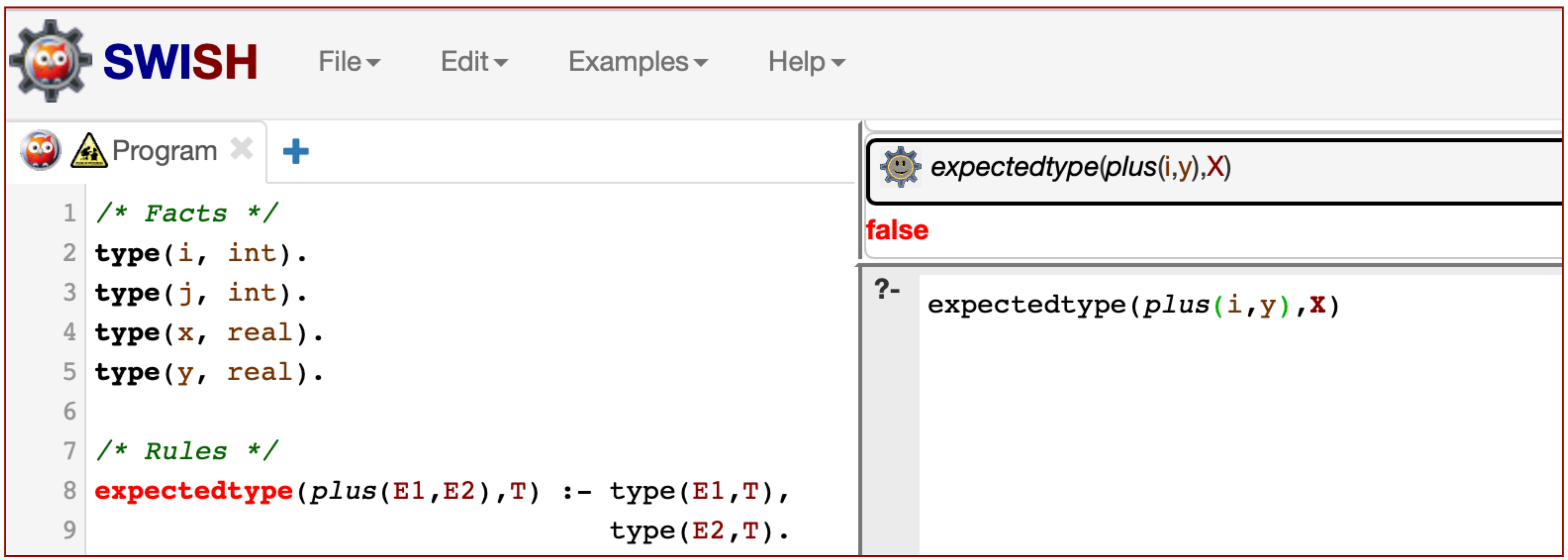
```prolog
?- expectedtype(plus(i,j),X)
```

Type Systems

Addition with Scope Context

$$\frac{S \vdash e_1 : T \qquad S \vdash e_2 : T \qquad T \text{ is a primitive type}}{S \vdash e_1 + e_2 : T}$$

Implementation in Prolog



Type Systems

Type Equivalence and Compatibility

What does it mean to say that two variable/values are equivalent?

$$1 \stackrel{?}{=} 1.0$$

$$1.0 \stackrel{?}{=} 1.000$$

$$\text{“c”} \stackrel{?}{=} \text{‘c’}$$

There are two approaches:

Name Equivalence
Types are equivalent if they have the same name.
I.e., they are the same if the programmer says they are the same.
Restrictive, but easier to implement than structural equivalence.

Structural Equivalence
Types are equivalent if they have the same structure.
I.e., they are the same if they are built the same: **same parts** in the **same order**.
Flexible, but harder to implement than name equivalence.

Type Equivalence and Compatibility

Name Equivalence
Types are equivalent if they have the same name.

first and *last* are the same type.
head and *tail* are the same type.
first and *head* are different types.

```
type link = ↑cell;

var  first : link;
     last  : link;
     head  : ↑cell;
     tail  : ↑cell;
```

Structural Equivalence
Types are equivalent if they have the same structure.

first, *last*, *head*, and *tail* are all the same type.

Type Equivalence and Compatibility

Name Equivalence

Types are equivalent if they have the same name.

MyRec and *YourRec* are different types.
a1, *a2*, and *a3* are all different types.

```
val MyRec   = { a=1, b=2 };
val YourRec = { a=1, b=2 };

var a1 = array[1..10] of int;
var a2 = array[1..2*5] of int;
var a3 = array[0..9] of int;
```

Structural Equivalence

Types are equivalent if they have the same structure.

MyRec and *YourRec* are the same type.
a1, *a2*, and *a3* are all the same type.