Universiteit Utrecht
[Faculty of Science
Information and Computing Sciences]

Compiler Construction

WWW: `http://www.cs.uu.nl/wiki/Cco`

Edition 2011/2012

Agenda

Simple types

- Syntax
- Semantics
- Type system
- Implementation

4. Simple types

Boolean expressions

§4

We extend our language of simple arithmetic expressions with support for Boolean expressions.

4.1 Syntax

Terms and values

n	$\in$	$\mathbf{Num}$	numerals
t	$\in$	$\mathbf{Tm}$	terms
v	$\in$	$\mathbf{Val}$	values

$$
\begin{array}{lll}
t & ::= & n \mid \mathtt{false} \mid \mathtt{true} \mid \mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3\ \mathbf{fi} \\
 & \mid & t_1 + t_2 \mid t_1 * t_2 \mid t_1 < t_2 \mid t_1 \equiv t_2 \mid t_2 > t_2 \\
v & ::= & n \mid \mathtt{false} \mid \mathtt{true}
\end{array}
$$

For example:

$$
\begin{array}{l}
\mathbf{if}\ 2 < 3 \\
\mathbf{then} \\
\quad \mathbf{if}\ \mathtt{false}\ \mathbf{then}\ 5\ \mathbf{else}\ 7\ \mathbf{fi} \\
\mathbf{else} \\
\quad \mathbf{if}\ 11 \equiv 13\ \mathbf{then}\ 17 * 19\ \mathbf{else}\ 23 + 29\ \mathbf{fi} \\
\mathbf{fi}
\end{array}
$$

4.2 Semantics

Preliminaries

Recall that $\mathbf{Num} = \mathbb{N}$.

We write $\sqsubset$ and $\sqsupset$ for the binary relations “less than” and “greater then” on natural numbers.

Small-step operational semantics

The semantics for our extended language is given as a small-step operational semantics, i.e., as a set of inference rules for deriving judgements of the form

$t \longrightarrow t'$	reduction

Additional and multiplication

$$\frac{}{n_1 + n_2 \longrightarrow n_1 \underline{+} n_2}\;[\textit{r-add}]$$

$$\frac{t_1 \longrightarrow t_1'}{t_1 + t_2 \longrightarrow t_1' + t_2}\;[\textit{r-add}_1]$$

$$\frac{t_2 \longrightarrow t_2'}{v_1 + t_2 \longrightarrow v_1 + t_2'}\;[\textit{r-add}_2]$$

$$\frac{}{n_1 * n_2 \longrightarrow n_1 \underline{*} n_2}\;[\textit{r-mul}]$$

$$\frac{t_1 \longrightarrow t_1'}{t_1 * t_2 \longrightarrow t_1' * t_2}\;[\textit{r-mul}_1]$$

$$\frac{t_2 \longrightarrow t_2'}{v_1 * t_2 \longrightarrow v_1 * t_2'}\;[\textit{r-mul}_2]$$

Less than

$$\frac{n_1 \sqsubset n_2}{n_1 < n_2 \longrightarrow \texttt{true}}\;[\textit{r-lt-true}]$$

$$\frac{n_1 \not\sqsubset n_2}{n_1 < n_2 \longrightarrow \texttt{false}}\;[\textit{r-lt-false}]$$

$$\frac{t_1 \longrightarrow t_1'}{t_1 < t_2 \longrightarrow t_1' < t_2}\;[\textit{r-lt}_1]$$

$$\frac{t_2 \longrightarrow t_2'}{v_1 < t_2 \longrightarrow v_1 < t_2'}\;[\textit{r-lt}_2]$$

☞ Two “computational” rules, two congruence rules.

Equals

$$\frac{n_1 = n_2}{n_1 \equiv n_2 \longrightarrow \texttt{true}}\;[\textit{r-eq-true}]$$

$$\frac{n_1 \neq n_2}{n_1 \equiv n_2 \longrightarrow \texttt{false}}\;[\textit{r-eq-false}]$$

$$\frac{t_1 \longrightarrow t_1'}{t_1 \equiv t_2 \longrightarrow t_1' \equiv t_2}\;[\textit{r-eq}_1]$$

$$\frac{t_2 \longrightarrow t_2'}{v_1 \equiv t_2 \longrightarrow v_1 \equiv t_2'}\;[\textit{r-eq}_2]$$

Greater than

$$\frac{n_1 \sqsupset n_2}{n_1 > n_2 \longrightarrow \texttt{true}}\ [\textit{r-gt-true}]$$

$$\frac{n_1 \not\sqsupset n_2}{n_1 > n_2 \longrightarrow \texttt{false}}\ [\textit{r-gt-false}]$$

$$\frac{t_1 \longrightarrow t_1'}{t_1 > t_2 \longrightarrow t_1' > t_2}\ [\textit{r-gt}_1]$$

$$\frac{t_2 \longrightarrow t_2'}{v_1 > t_2 \longrightarrow v_1 > t_2'}\ [\textit{r-gt}_2]$$

Conditionals

$$\frac{}{\textbf{if}\ \texttt{true}\ \textbf{then}\ t_2\ \textbf{else}\ t_3\ \textbf{fi} \longrightarrow t_2}\ [\textit{r-if-true}]$$

$$\frac{}{\textbf{if}\ \texttt{false}\ \textbf{then}\ t_2\ \textbf{else}\ t_3\ \textbf{fi} \longrightarrow t_3}\ [\textit{r-if-false}]$$

$$\frac{t_1 \longrightarrow t_1'}{\textbf{if}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3\ \textbf{fi} \longrightarrow \textbf{if}\ t_1'\ \textbf{then}\ t_2\ \textbf{else}\ t_3\ \textbf{fi}}\ [\textit{r-if}]$$

☞ Conditionals are nonstrict: one branch is always left unevaluated.

Example

$$\begin{array}{l}
\textbf{if}\ 2+1<3\ \textbf{then}\ 5*7\ \textbf{else}\ 11*13\ \textbf{fi}\\
\longrightarrow \textbf{if}\ 3<3\ \textbf{then}\ 5*7\ \textbf{else}\ 11*13\ \textbf{fi}\\
\longrightarrow \textbf{if}\ \texttt{false}\ \textbf{then}\ 5*7\ \textbf{else}\ 11*13\ \textbf{fi}\\
\longrightarrow 11*13\\
\longrightarrow 143
\end{array}$$

Not all terms evaluate to a value

$$\begin{array}{l}
\textbf{if}\ 3<3\ \textbf{then}\ 5\ \textbf{else}\ \texttt{true}+7\ \textbf{fi}\\
\longrightarrow \textbf{if}\ \texttt{false}\ \textbf{then}\ 5\ \textbf{else}\ \texttt{true}+7\ \textbf{fi}\\
\longrightarrow \texttt{true}+7
\end{array}$$

None of the reduction rules applies to $\texttt{true} + 7$.

Still, $\texttt{true} + 7$ is not a value!

Possible "solutions"

- Leave it as it is: some programs are literally "meaningless". Implementations will produce a run-time error if an irreducible term that is not a value is encountered.
- Extend the language of values such that terms like $\texttt{true} + 7$ are considered values and hence denote possible meanings of programs.
- Extend the operational semantics so that terms like $\texttt{true} + 7$ can actually be reduced, for example by performing coercions between natural numbers and Boolean values.
- Do not consider (some classes of) irreducible terms as programs.

4.3 Type system

Type systems

"A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute."

—Pierce (*TAPL*)

Type systems provide a means to statically (i.e., at compile-time, without running the program) calculate an approximation of the run-time behaviour of a program.

☞ Sometimes we speak of "dynamically typed languages" to refer to languages in which run-time tags are used to distinguish between different kinds of data.

What are type systems good for?

- (Strong) type systems guarantee the absence of certain sorts of run-time errors.
- Type systems allow programmers to think about their programs on a more abstract level.
- Type systems provide machine-checkable documentation for programs.
- Type systems protect the abstractions they provide.
- Type systems enable certain program optimisations: compilers can take advantage of the fact that the run-time form of values is known at compile-time.

Small-step operational semantics, revisited

Recall: a small-step (or structured) operational semantics formalises the notion of a single computational step (or reduction) in the evaluation of a term:

$t \longrightarrow t'$ reduction

The complete evaluation of a term can be formalised by considering chains $t_1 \longrightarrow^* t_n$ of reduction steps:

$$t_1 \longrightarrow t_2 \longrightarrow \cdots \longrightarrow t_{n-1} \longrightarrow t_n$$

☞ The multi-step reduction relation $\longrightarrow^*$ on terms is the reflexive, transitive closure of the single-step reduction relation $\longrightarrow$: **(1)** for all t and t', if $t \longrightarrow t'$, then $t \longrightarrow^* t'$ (a singleton chain); **(2)** for all t, $t \longrightarrow^* t$ (the empty chain); and **(3)** for all t, t', t'', if $t \longrightarrow^* t'$ and $t' \longrightarrow^* t''$, then $t \longrightarrow^* t''$ (chain concatenation).

Normal forms and stuck terms

An irreducible term (i.e., a term t for which there is no t' such that $t \longrightarrow t'$) is called a normal form.

If $t \longrightarrow^* t'$ and t' is a normal form, then we say that t' is a normal form of t.

A normal form that is not a value (i.e., a normal form t with $t \notin \mathbf{Val}$) is called a stuck term.

☞ So, a normal form is either a stuck term or a value.

Stuck terms: example

$$\begin{array}{l}
\mathbf{if}\ 2+1<3\ \mathbf{then}\ 5\ \mathbf{else}\ \mathtt{true}+7\ \mathbf{fi} \\
\longrightarrow \mathbf{if}\ 3<3\ \mathbf{then}\ 5\ \mathbf{else}\ \mathtt{true}+7\ \mathbf{fi} \\
\longrightarrow \mathbf{if}\ \mathtt{false}\ \mathbf{then}\ 5\ \mathbf{else}\ \mathtt{true}+7\ \mathbf{fi} \\
\longrightarrow \mathtt{true}+7 \\
\not\longrightarrow
\end{array}$$

We have

$$\mathbf{if}\ 2+1<3\ \mathbf{then}\ 5\ \mathbf{else}\ \mathtt{true}+7\ \mathbf{fi} \longrightarrow^* \mathtt{true}+7$$

where the normal form $\mathtt{true}+7$ is stuck.

Stuck terms and run-time errors

Intuitively, stuck terms result from evaluating erroneous programs. ("You cannot add numbers to Booleans.")

Stuck terms in an operational semantics then correspond to the notion of a run-time error: in a concrete representation such an error may be witnessed as, for example, an exception or a segmentation fault.

Types

Stuck terms corresponding to erroneous programs, we would like to tell, without evaluating a term, that it will not get stuck.

Approach: we distinguish between terms that result in numbers and terms that result in Boolean values.

So, we introduce two types: Nat and $Bool$.

$\tau \in \mathbf{Ty}$ types

$\tau ::= Nat \mid Bool$

☞ $Bool$ is a type both in our object language (simple arithmetic and boolean expressions) and in our implementation language (Haskell).

Typing

Next, we classify terms according to their type.

We define a natural deduction system to establish judgements of the form

$t : \tau$ typing

meaning that a term t has type τ, that is, without actually evaluating t, we can determine that it will evaluate to a value of the appropriate form.

Values belonging to the type Nat of natural numbers are the numerals n.

The values that belong to the type $Bool$ are the constants `false` and `true`.

Typing: constants

Numerals n and the constants `false` and `true` are already values: their types follow immediately.

$$\frac{}{n : Nat}\ [\textit{t-num}]$$

$$\frac{}{\texttt{false} : Bool}\ [\textit{t-false}]$$

$$\frac{}{\texttt{true} : Bool}\ [\textit{t-true}]$$

Typing: conditionals

Conditionals (terms of the form $\mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3\ \mathbf{fi}$) are not normal forms.

Evaluation can only proceed (cf. rule [*r-if*]) towards a normal form if t_1 evaluates (in zero or more steps) to `false` (rule [*r-if-false*]) or `true` (rule [*r-if-true*]), i.e. a value of type $Bool$.

To determine that the conditional evaluates to a value of some type τ, both branches t_2 and t_3 must evaluate to a value of that same type τ.

$$\frac{t_1 : Bool \quad t_2 : \tau \quad t_3 : \tau}{\mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3\ \mathbf{fi} : \tau}\ [\textit{t-if}]$$

☞ The three uses of the single metavariable τ denote the constraint that the type assigned to the conditional must be the same as that of both the **then**- and the **else**-branch.

Typing: arithmetic operators

If both operands of an addition or multiplication evaluate to a natural number, then the arithmetic operation will evaluate to a number.

$$\frac{t_1 : Nat \quad t_2 : Nat}{t_1 + t_2 : Nat}\ [\textit{t-add}]$$

$$\frac{t_1 : Nat \quad t_2 : Nat}{t_1 * t_2 : Nat}\ [\textit{t-mul}]$$

Typing: relational operators

If both operands of a relational operator evaluate to a natural number, then the comparison will evaluate to a Boolean value.

$$\frac{t_1 : Nat \quad t_2 : Nat}{t_1 < t_2 : Bool}\ [\textit{t-lt}]$$

$$\frac{t_1 : Nat \quad t_2 : Nat}{t_1 \equiv t_2 : Bool}\ [\textit{t-eq}]$$

$$\frac{t_1 : Nat \quad t_2 : Nat}{t_1 > t_2 : Bool}\ [\textit{t-gt}]$$

Typing: example

$$\textbf{if } 2+1<3 \textbf{ then } 5*7 \textbf{ else } 11*13 \textbf{ fi} : Nat$$

Inference tree:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{2 : Nat} \quad \dfrac{}{1 : Nat}}{2+1 : Nat} \quad \dfrac{}{3 : Nat}}{2+1<3 : Bool} \quad \dfrac{\dfrac{}{5 : Nat} \quad \dfrac{}{7 : Nat}}{5*7 : Nat} \quad \dfrac{\dfrac{}{11 : Nat} \quad \dfrac{}{13 : Nat}}{11*13 : Nat}}{\textbf{if } 2+1<3 \textbf{ then } 5*7 \textbf{ else } 11*13 \textbf{ fi} : Nat}$$

Well- and ill-typed terms

If there exist a type τ such that $t : \tau$, then t is a well-typed term.

Similarly, if there does not exist a type τ for which it can be established that $t : \tau$, then t is an ill-typed term.

Ill-typed terms: example

There is no type τ such that

$$\mathbf{if}\ 2 + 1 < 3\ \mathbf{then}\ 5\ \mathbf{else}\ \mathtt{true} + 7\ \mathbf{fi} : \tau$$

Hence, the term $\mathbf{if}\ 2 + 1 < 3\ \mathbf{then}\ 5\ \mathbf{else}\ \mathtt{true} + 7\ \mathbf{fi}$ is ill-typed.

☞ The ill-typedness of a term does not follow from a single rule: it is the *lacking* of some *extra*, suitable rule that makes that we cannot assign a type to the example term above.

Type safety

Well-typed terms share two important properties:

1. **Progress:** A well-typed term t is never stuck, i.e., either t is a value or else there exists a term t' with $t \longrightarrow t'$.
2. **Preservation:** If t is a well-typed term and t' is a term with with $t \longrightarrow t'$, then t' is also well-typed. (Often, but not always, t and t' even have the *same* type.)

Together, progress and preservation establish type safety: a normal form of a well-typed term is never stuck (Wright and Felleisen, *Inf. Comput. 115*).

☞ Preservation is sometimes also referred to as *subject reduction* or *subject evaluation*.

Static vs. dynamic semantics

A type system provides a static semantics for a programming language: a way to assign a meaning to a program without actually running it.

To contrast it with a static semantics, the formalisation of the evaluation of a program (for example, by means of a set of reduction rules) is called a dynamic semantics.

The meaning (i.e., the type) assigned to a program by a static semantics is an abstraction of the dynamic semantics (i.e., the value) of the program.

For example: the type Bool abstracts away from the concrete values `false` and `true`.

Soundness

We consider a type system correct if it makes accurate predictions of the form of the values a term evaluates to:

A type system is **sound** with respect to the operational semantics if for every well-typed term t, $t : \tau$ and $t \longrightarrow^* v$ imply that $v : \tau$.

☞ Well-typedness of a term does not imply that it actually does evaluate to a value. In a more involved language there may be nonterminating terms that do not have any normal form at all.

☞ A type system is generally not *complete* w.r.t. the operational semantics: $t \longrightarrow^* v$ and $v : \tau$ do not imply that $t : \tau$.

Conservativeness

Being static, type systems typically need to be conservative: even though some terms do evaluate to values, they are considered ill-typed.

For example:

if `false` **then** 0 **else** `true` **fi**

evaluates to `true` but cannot be assigned a type.

☞ In the example above, it is of course possible to statically predict that the term will evaluate to a Boolean value. In general, however, and for less trival languages, the guard of a conditional can be an arbitrary and potentially nonterminating expression.

Algorithms

An algorithm for assigning types to terms should fulfill two fundamental requirements:

1. **Soundness:** If the algorithm assigns a type τ to a term t, then we can in fact establish, by means of the typing rules, that $t : \tau$.
2. **Completeness:** The algorithm should assign a type to every well-typed term.

As a result, a correct algorithm should fail assign a type to an ill-typed term.

☞ Soundness and completeness of a typing algorithm with respect to a type system are sometimes also referred to as *syntactic soundness* and *syntactic completeness* (as a means to distinguish syntactic soundness from semantic soundness, i.e., soundness w.r.t. a dynamic semantics).

4.4 Implementation

An interpreter with simple types

We now construct an interpreter for our simple language of arithmetic and Boolean expressions.

To our interpreter, an input is a (valid) program, if it is both a syntactically correct and well-typed term.

The components it consists of are:

- A parser.
- A type checker.
- An evaluator.
- A pretty printer.

Syntax: numerals, terms, and values


```
type Num_ = Integer
data Tm   = Num Num_ | False_ | True_
          | If Tm Tm Tm
          | Add Tm Tm | Mul Tm Tm
          | Lt Tm Tm | Eq Tm Tm | Gt Tm Tm
data Val  = VNum Num_ | VFalse | VTrue
```

☞ We include underscore postfixes in the constructor names *False_* and *True_* to distinguish them from the *Bool*-constructors *False* and *True*.

Alternatively, we could represent them by a single constructor *Bool* : *Bool* → *Tm* that delegates to the *Prelude*-type *Bool*.

Syntax: numerals, terms, and values (cont'd)


```
instance Tree Tm where
  fromTree = fromTree_Tm
  toTree   = toTree_Tm
```


Syntax: numerals, terms, and values (cont'd)


```
fromTree_Tm :: Tm → ATerm
fromTree_Tm (Num n)      = App "Num"   [fromTree n]
fromTree_Tm False_       = App "False" []
fromTree_Tm True_        = App "True"  []
fromTree_Tm (If t1 t2 t3) = App "If"   [fromTree t1, fromTree t2,
                                        fromTree t3]
fromTree_Tm (Add t1 t2)  = App "Add"   [fromTree t1, fromTree t2]
fromTree_Tm (Mul t1 t2)  = App "Mul"   [fromTree t1, fromTree t2]
fromTree_Tm (Lt t1 t2)   = App "Lt"    [fromTree t1, fromTree t2]
fromTree_Tm (Eq t1 t2)   = App "Eq"    [fromTree t1, fromTree t2]
fromTree_Tm (Gt t1 t2)   = App "Gt"    [fromTree t1, fromTree t2]
```


Syntax: numerals, terms, and values (cont'd)


```
toTree_Tm :: ATerm → Feedback Tm
toTree_Tm = parseTree
              [ app "Num"   (Num <$> arg)
              , app "False" (pure False_)
              , app "True"  (pure True_)
              , app "If"    (If <$> arg <*> arg <*> arg)
              , app "Add"   (Add <$> arg <*> arg)
              , app "Mul"   (Mul <$> arg <*> arg)
              , app "Lt"    (Lt <$> arg <*> arg)
              , app "Eq"    (Eq <$> arg <*> arg)
              , app "Gt"    (Gt <$> arg <*> arg)
              ]
```

Syntax: numerals, terms, and values (cont'd)


```
instance Tree Val where
  fromTree (VNum n) = App "VNum"   [fromTree n]
  fromTree VFalse   = App "VFalse" []
  fromTree VTrue    = App "VTrue"  []
  toTree = parseTree [app "VNum"   (VNum <$> arg)
                     , app "VFalse" (pure VFalse)
                     , app "VTrue"  (pure VTrue)
                     ]
```


Syntax: types


```
data Ty = Nat | Bool deriving Eq
```

```
instance Tree Ty where
  fromTree Nat  = App "Nat"  []
  fromTree Bool = App "Bool" []
  toTree = parseTree [app "Nat"  (pure Nat)
                     , app "Bool" (pure Bool)
                     ]
```

☞ We make *Ty* an instance of *Eq*, so we can test two types for syntactic equality.

Evaluation

We implement the operational semantics by means of a partial function from terms to values:

```
eval : Tm → Val
```


Evaluation: constants


```
eval (Num n) = VNum n
eval False_  = VFalse
eval True_   = VTrue
```

Evaluation: conditionals

$$
\begin{aligned}
\mathit{eval}\ (\mathit{If}\ t_1\ t_2\ t_3) = \mathbf{case}\ &\mathit{eval}\ t_1\ \mathbf{of} \\
&\mathit{VTrue} \rightarrow \mathit{eval}\ t_2 \\
&\mathit{VFalse} \rightarrow \mathit{eval}\ t_3
\end{aligned}
$$

☞ The evaluation function fails if first subterms of a conditional does not not evaluate to a VTrue or VFalse.

Evaluation: arithmetic operators

$$
\begin{aligned}
\mathit{eval}\ (\mathit{Add}\ t_1\ t_2) = \mathbf{let}\ &\mathit{VNum}\ n_1 = \mathit{eval}\ t_1 \\
&\mathit{VNum}\ n_2 = \mathit{eval}\ t_2 \\
\mathbf{in}\ &\mathit{VNum}\ (n_1 + n_2) \\
\mathit{eval}\ (\mathit{Mul}\ t_1\ t_2) = \mathbf{let}\ &\mathit{VNum}\ n_1 = \mathit{eval}\ t_1 \\
&\mathit{VNum}\ n_2 = \mathit{eval}\ t_2 \\
\mathbf{in}\ &\mathit{VNum}\ (n_1 * n_2)
\end{aligned}
$$

☞ The evaluation function fails if one of the subterms of an arithmetic operator does not evaluate to a VNum-value.

Evaluation: relational operators

$$
\begin{aligned}
\mathit{eval}\ (\mathit{Lt}\ t_1\ t_2) = \mathbf{let}\ &\mathit{VNum}\ n_1 = \mathit{eval}\ t_1 \\
&\mathit{VNum}\ n_2 = \mathit{eval}\ t_2 \\
\mathbf{in}\ &\mathbf{if}\ n_1 < n_2\ \mathbf{then}\ \mathit{VTrue}\ \mathbf{else}\ \mathit{VFalse} \\
\mathit{eval}\ (\mathit{Eq}\ t_1\ t_2) = \mathbf{let}\ &\mathit{VNum}\ n_1 = \mathit{eval}\ t_1 \\
&\mathit{VNum}\ n_2 = \mathit{eval}\ t_2 \\
\mathbf{in}\ &\mathbf{if}\ n_1 \equiv n_2\ \mathbf{then}\ \mathit{VTrue}\ \mathbf{else}\ \mathit{VFalse} \\
\mathit{eval}\ (\mathit{Gt}\ t_1\ t_2) = \mathbf{let}\ &\mathit{VNum}\ n_1 = \mathit{eval}\ t_1 \\
&\mathit{VNum}\ n_2 = \mathit{eval}\ t_2 \\
\mathbf{in}\ &\mathbf{if}\ n_1 > n_2\ \mathbf{then}\ \mathit{VTrue}\ \mathbf{else}\ \mathit{VFalse}
\end{aligned}
$$

☞ The evaluation function fails if one of the subterms of a comparison does not evaluate to a VNum-value.

Evaluation: assessment

If the evaluation function encounters a stuck term, it fails with a run-time error and hence, the interpreter crashes—which may be a bit harsh on the user.

We could of course have the evaluation function run inside the *Feedback*-monad, but while this perhaps suits an interpreter, an arguably better option for a compiler may be to employ a static semantics and produce from within an implementation of a type system helpful type-error messages for ill-typed programs.

Even for interpreters, this approach establishes, to some extent, a separation of concerns in the implementation.

Typing

We implement the static semantics by means of a typing function that runs inside the *Feedback*-monad and that tries to assign types to terms:

$$typeOf :: Tm \to Feedback\ Ty$$

Typing: constants


```
typeOf (Num n) = return Nat
typeOf False_  = return Bool
typeOf True_   = return Bool
```


Typing: conditionals


```
typeOf (If t1 t2 t3) =
  do τ1 ← typeOf t1
     case τ1 of
       Bool → do τ2 ← typeOf t2
                 τ3 ← typeOf t3
                 if τ2 ≡ τ3
                   then return τ2
                   else fail $ "arms of conditional " ++
                               "have different types"
       _ → fail "guard of conditional is not a Boolean"
```

☞ The actual error messages may provide more detail.

Typing: arithmetic operators


```
typeOf (Add t1 t2) =
  do τ1 ← typeOf t1
     τ2 ← typeOf t2
     case (τ1, τ2) of
       (Nat, Nat) → return Nat
       _ → fail $ "operands of addition are " ++
                  "not both numbers"
typeOf (Mul t1 t2) =
  do τ1 ← typeOf t1
     τ2 ← typeOf t2
     case (τ1, τ2) of
       (Nat, Nat) → return Nat
       _ → fail $ "operands of multiplication are " ++
                  "not both numbers"
```

Typing: relational operators

$$
\begin{array}{l}
\mathit{typeOf}\ (\mathit{Lt}\ t_1\ t_2) = \\
\quad \mathbf{do}\ \tau_1 \leftarrow \mathit{typeOf}\ t_1 \\
\qquad \tau_2 \leftarrow \mathit{typeOf}\ t_2 \\
\qquad \mathbf{case}\ (\tau_1, \tau_2)\ \mathbf{of} \\
\qquad\quad (\mathit{Nat}, \mathit{Nat}) \rightarrow \mathit{return}\ \mathit{Bool} \\
\qquad\quad \_ \rightarrow \mathit{fail}\ \texttt{"operands of comparison are not both numbers"} \\
\mathit{typeOf}\ (\mathit{Eq}\ t_1\ t_2) = \\
\quad \mathbf{do}\ \tau_1 \leftarrow \mathit{typeOf}\ t_1 \\
\qquad \tau_2 \leftarrow \mathit{typeOf}\ t_2 \\
\qquad \mathbf{case}\ (\tau_1, \tau_2)\ \mathbf{of} \\
\qquad\quad (\mathit{Nat}, \mathit{Nat}) \rightarrow \mathit{return}\ \mathit{Bool} \\
\qquad\quad \_ \rightarrow \mathit{fail}\ \texttt{"operands of comparison are not both numbers"} \\
\mathit{typeOf}\ (\mathit{Gt}\ t_1\ t_2) = \\
\quad \mathbf{do}\ \tau_1 \leftarrow \mathit{typeOf}\ t_1 \\
\qquad \tau_2 \leftarrow \mathit{typeOf}\ t_2 \\
\qquad \mathbf{case}\ (\tau_1, \tau_2)\ \mathbf{of} \\
\qquad\quad (\mathit{Nat}, \mathit{Nat}) \rightarrow \mathit{return}\ \mathit{Bool} \\
\qquad\quad \_ \rightarrow \mathit{fail}\ \texttt{"operands of comparison are not both numbers"}
\end{array}
$$