Issues in the Semantics of Programming Languages

Lecture 9

Objectives

By the end of this lecture you should be able to:

1. Construct three-address code.
2. Construct SDDs/SDTs for declarations.
3. Construct SDDs/SDTs for type conversion.
4. Construct SDDs/SDTs for overloaded operators.
5. Infer the type of an expression.

Outline

Outline

An Intermediate Representation

- A source program is often translated into an intermediate three-address code.

Structure of Three-Address Code

- A sequence of instructions.
- Each instruction is
  1. a (conditional or unconditional) jump;
  2. has exactly one operator; or
  3. has exactly two operators, one of them an assignment.
- Each instruction has at most three addresses.
- An address is either (i) a name (identifier), a constant, or a compiler-generated temporary.

An Intermediate Representation

- A source program is often translated into an intermediate three-address code.

Structure of Three-Address Code

- A sequence of instructions.
- Each instruction is
  1. a (conditional or unconditional) jump;
  2. has exactly one operator; or
  3. has exactly two operators, one of them an assignment.
- Each instruction has at most three addresses.
- An address is either (i) a name (identifier), a constant, or a compiler-generated temporary.

An Intermediate Representation

- A source program is often translated into an intermediate three-address code.

Structure of Three-Address Code

- A sequence of instructions.
- Each instruction is
  1. a (conditional or unconditional) jump;
  2. has exactly one operator; or
  3. has exactly two operators, one of them an assignment.
- Each instruction has at most three addresses.
- An address is either (i) a name (identifier), a constant, or a compiler-generated temporary.

An Intermediate Representation

- A source program is often translated into an intermediate three-address code.

Structure of Three-Address Code

- A sequence of instructions.
- Each instruction is
  1. a (conditional or unconditional) jump;
  2. has exactly one operator; or
  3. has exactly two operators, one of them an assignment.
- Each instruction has at most three addresses.
- An address is either (i) a name (identifier), a constant, or a compiler-generated temporary.

An Intermediate Representation

- A source program is often translated into an intermediate three-address code.

Structure of Three-Address Code

- A sequence of instructions.
- Each instruction is
  1. a (conditional or unconditional) jump;
  2. has exactly one operator; or
  3. has exactly two operators, one of them an assignment.
- Each instruction has at most three addresses.
- An address is either (i) a name (identifier), a constant, or a compiler-generated temporary.

An Intermediate Representation

- A source program is often translated into an intermediate three-address code.

Structure of Three-Address Code

- A sequence of instructions.
- Each instruction is
  1. a (conditional or unconditional) jump;
  2. has exactly one operator; or
  3. has exactly two operators, one of them an assignment.
- Each instruction has at most three addresses.
- An address is either (i) a name (identifier), a constant, or a compiler-generated temporary.

An Intermediate Representation

- A source program is often translated into an intermediate three-address code.

Structure of Three-Address Code

- A sequence of instructions.
- Each instruction is
  1. a (conditional or unconditional) jump;
  2. has exactly one operator; or
  3. has exactly two operators, one of them an assignment.
- Each instruction has at most three addresses.
- An address is either (i) a name (identifier), a constant, or a compiler-generated temporary.

An Intermediate Representation

- A source program is often translated into an intermediate three-address code.

Structure of Three-Address Code

- A sequence of instructions.
- Each instruction is
  1. a (conditional or unconditional) jump;
  2. has exactly one operator; or
  3. has exactly two operators, one of them an assignment.
- Each instruction has at most three addresses.
- An address is either (i) a name (identifier), a constant, or a compiler-generated temporary.

An Intermediate Representation

- A source program is often translated into an intermediate three-address code.

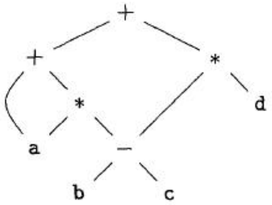
Structure of Three-Address Code

- A sequence of instructions.
- Each instruction is
  1. a (conditional or unconditional) jump;
  2. has exactly one operator; or
  3. has exactly two operators, one of them an assignment.
- Each instruction has at most three addresses.
- An address is either (i) a name (identifier), a constant, or a compiler-generated temporary.

Simple Arithmetic Expression

Example

An expression DAG for `a + a * (b-c) + (b-c) * d`



$$
\begin{aligned}
t_1 &= b - c\\
t_2 &= a * t_1\\
t_3 &= a + t_2\\
t_4 &= t_1 * d\\
t_5 &= t_3 + t_4
\end{aligned}
$$

©Aho et al. (2007)

While Loop

Example

Source Code

```
do i = i + 1;
while(a[i] < v)
```

Three-Address Code

```
100:  t1 = i + 1
101:  i = t1
102:  t2 = i * 8
103:  t3 = a[t2]
104:  t4 = t3 < v
105:  if t4 goto 100
```

Outline

1. Three-Address Code
2. Types and Declarations
3. Type Checking

Types

- Data types are classes of data.
- Each with a set of meaningful operations.
- Since such classes are typically infinite, types are represented by type expressions.
- Each expression is either a basic type or is formed by applying a type constructor to a type expression.

Types

- Data types are classes of data.
- Each with a set of meaningful operations.
- Since such classes are typically infinite, types are represented by type expressions.
- Each expression is either a basic type or is formed by applying a type constructor to a type expression.

Types

- Data types are classes of data.
- Each with a set of meaningful operations.
- Since such classes are typically infinite, types are represented by type expressions.
- Each expression is either a basic type or is formed by applying a type constructor to a type expression.

Types

- Data types are classes of data.
- Each with a set of meaningful operations.
- Since such classes are typically infinite, types are represented by type expressions.
- Each expression is either a basic type or is formed by applying a type constructor to a type expression.

Type Expressions

We consider the following common type expressions.

1. A basic type (e.g. `int`, `float`, `char`, `void`)
2. A type name (e.g `data Bit = O | I`)
3. A type variable
4. **record** $\{e1\ id_1, e_2\ id_2, \ldots, e_n\ id_n\}$
5. $e\ [num]$
6. $e_1 \longrightarrow e_2$
7. $e_1 \times e_2$

Type Expressions

We consider the following common type expressions.

1. A basic type (e.g. `int`, `float`, `char`, `void`)
2. A type name (e.g `data Bit = O | I`)
3. A type variable
4. **record** $\{e1\ id_1, e_2\ id_2, \ldots, e_n\ id_n\}$
5. $e\ [num]$
6. $e_1 \longrightarrow e_2$
7. $e_1 \times e_2$

Type Expressions

We consider the following common type expressions.

1. A basic type (e.g. `int`, `float`, `char`, `void`)
2. A type name (e.g `data Bit = O | I`)
3. A type variable
4. **record** $\{e1\ id_1, e_2\ id_2, \ldots, e_n\ id_n\}$
5. $e\ [num]$
6. $e_1 \longrightarrow e_2$
7. $e_1 \times e_2$

Type Expressions

We consider the following common type expressions.

1. A basic type (e.g. `int`, `float`, `char`, `void`)
2. A type name (e.g `data Bit = O | I`)
3. A type variable
4. **record** $\{e1\ id_1, e_2\ id_2, \ldots, e_n\ id_n\}$
5. $e\ [num]$
6. $e_1 \longrightarrow e_2$
7. $e_1 \times e_2$

Type Expressions

We consider the following common type expressions.

1. A basic type (e.g. `int`, `float`, `char`, `void`)
2. A type name (e.g `data Bit = O | I`)
3. A type variable
4. **record** $\{e1\ id_1, e_2\ id_2, \ldots, e_n\ id_n\}$
5. $e\ [num]$
6. $e_1 \longrightarrow e_2$
7. $e_1 \times e_2$

Type Expressions

We consider the following common type expressions.

1. A basic type (e.g. `int`, `float`, `char`, `void`)
2. A type name (e.g `data Bit = O | I`)
3. A type variable
4. **record** $\{e1\ id_1, e_2\ id_2, \ldots, e_n\ id_n\}$
5. $e\ [num]$
6. $e_1 \longrightarrow e_2$
7. $e_1 \times e_2$

Type Expressions

We consider the following common type expressions.

1. A basic type (e.g. `int`, `float`, `char`, `void`)
2. A type name (e.g `data Bit = O | I`)
3. A type variable
4. **record** $\{e1\ id_1, e_2\ id_2, \ldots, e_n\ id_n\}$
5. $e\ [num]$
6. $e_1 \longrightarrow e_2$
7. $e_1 \times e_2$

Declarations

Example (Java-like Declarations)

$$
\begin{array}{lcl}
D & \longrightarrow & T\ \textbf{id};\ D \mid \varepsilon \\
T & \longrightarrow & BC \mid \textbf{record}\ \{\ D\ \} \\
B & \longrightarrow & \textbf{int} \mid \textbf{float} \\
C & \longrightarrow & [\textbf{num}]C \mid \varepsilon
\end{array}
$$

Type Width

- The number of storage units (typically, bytes) may be determined at compile-time for static data structures.
- In such cases, it is possible to determine, from the type expression of a data structure, the number of these storage units.
- This number is referred to as the type width.
- We assume that type width is an integral number of storage units.
- We also assume that said storage units are allocated contiguously.

SDD for Basic and Array Types

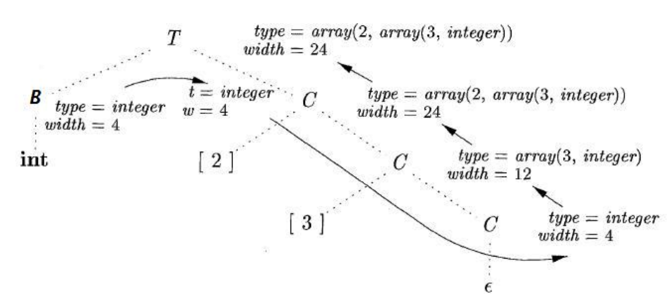
Example

$$
\begin{array}{llll}
T & \longrightarrow & BC & C.t = B.type; C.w = B.width \\
 & & & T.type = C.type; T.width = C.width \\
B & \longrightarrow & \textbf{int} & B.type = integer; B.width = 4 \\
B & \longrightarrow & \textbf{float} & B.type = float; B.width = 8 \\
C & \longrightarrow & [\textbf{num}]C_1 & C_1.t = C.t; C_1.w = C.w \\
 & & & C.type = array(\textbf{num}.value, C_1.type) \\
 & & & C.width = \textbf{num}.value \times C_1.width \\
C & \longrightarrow & \varepsilon & C.type = C.t; C.width = C.w
\end{array}
$$

Simple Array Expression

Example

Annotated parse tree for `int[2][3]`



©Aho et al. (2007)

Memory Allocation to Variables

- Physical memory allocation to variables is done at run-time by the OS.
- At compile-time, only relative addresses are assigned.
- Relative addresses are managed by maintaining a variable *offset* which carries the address of the next variable to be created relative to a virtual address space.

Memory Allocation to Variables

- Physical memory allocation to variables is done at run-time by the OS.
- At compile-time, only relative addresses are assigned.
- Relative addresses are managed by maintaining a variable *offset* which carries the address of the next variable to be created relative to a virtual address space.

Memory Allocation to Variables

- Physical memory allocation to variables is done at run-time by the OS.
- At compile-time, only relative addresses are assigned.
- Relative addresses are managed by maintaining a variable *offset* which carries the address of the next variable to be created relative to a virtual address space.

Assigning Relative Addresses

Example (SDT)

$$
\begin{array}{lcll}
P & \longrightarrow & \{offset = 0\}\ D & \\
D & \longrightarrow & T\ \mathbf{id}; & \{top.put(\mathbf{id}.lexeme, T.type, offset) \\
 & & & offset = offset + T.width\} \\
 & \longrightarrow & D_1 & \\
D & \longrightarrow & \varepsilon &
\end{array}
$$

Records

- Record types are implemented by objects of the form $record(t)$, where t is a symbol table.
- Said symbol table contains entries for each field of the record object.
- An entry contains the name, type, and relative address of a field.

Records

- Record types are implemented by objects of the form $record(t)$, where t is a symbol table.
- Said symbol table contains entries for each field of the record object.
- An entry contains the name, type, and relative address of a field.

Records

- Record types are implemented by objects of the form $record(t)$, where t is a symbol table.
- Said symbol table contains entries for each field of the record object.
- An entry contains the name, type, and relative address of a field.

Record SDT

Example

$$
\begin{array}{llll}
T & \longrightarrow & \textbf{record}\{ & \{Env.push(top); top = \textbf{new}\ Table() \\
 & & & Stack.push(offset); offset = 0\} \\
 & & D\} & \{T.type = record(top); T.width = offset \\
 & & & top = Env.pop(); offset = Stack.pop()\}
\end{array}
$$

Outline

1 Three-Address Code

2 Types and Declarations

3 Type Checking

What is Type Checking?

- Type checking consists in two steps:
  1. Assigning a type expression to each program component.
  2. Checking that these type expressions observe a number of logical rules constituting the type system of the language.
- If the target code carries the type of an expression along with its value, type checking can be done at run-time.
- An implementation of a language is strongly-typed if type-checking could be done at compile-time.

What is Type Checking?

- Type checking consists in two steps:
  1. Assigning a type expression to each program component.
  2. Checking that these type expressions observe a number of logical rules constituting the type system of the language.
- If the target code carries the type of an expression along with its value, type checking can be done at run-time.
- An implementation of a language is strongly-typed if type-checking could be done at compile-time.

What is Type Checking?

- Type checking consists in two steps:
  1. Assigning a type expression to each program component.
  2. Checking that these type expressions observe a number of logical rules constituting the type system of the language.
- If the target code carries the type of an expression along with its value, type checking can be done at run-time.
- An implementation of a language is strongly-typed if type-checking could be done at compile-time.

What is Type Checking?

- Type checking consists in two steps:
  1. Assigning a type expression to each program component.
  2. Checking that these type expressions observe a number of logical rules constituting the type system of the language.
- If the target code carries the type of an expression along with its value, type checking can be done at run-time.
- An implementation of a language is strongly-typed if type-checking could be done at compile-time.

What is Type Checking?

- Type checking consists in two steps:
  1. Assigning a type expression to each program component.
  2. Checking that these type expressions observe a number of logical rules constituting the type system of the language.
- If the target code carries the type of an expression along with its value, type checking can be done at run-time.
- An implementation of a language is strongly-typed if type-checking could be done at compile-time.

Type Synthesis and Inference

- Type checking can be done by either synthesis or inference.
- A rule for type synthesis builds up the type of an expression from the type of its sub-expressions.
  - This requires that identifiers have declared their types before use.

Example

if f has type $s \longrightarrow t$ **and** x has type s, **then** $f(x)$ has type t.

- A rule for type inference infers the type of an expression from the way it is used.

Example (α and β are type variables)

if $f(x)$ is an expression, **then** there are α and β such that $f(x)$ has type $\alpha \longrightarrow \beta$ **and** x has type α.

Type Synthesis and Inference

- Type checking can be done by either synthesis or inference.
- A rule for type synthesis builds up the type of an expression from the type of its sub-expressions.
  - This requires that identifiers have declared their types before use.

Example

if f has type $s \longrightarrow t$ **and** x has type s, **then** $f(x)$ has type t.

- A rule for type inference infers the type of an expression from the way it is used.

Example (α and β are type variables)

if $f(x)$ is an expression, **then** there are α and β such that $f(x)$ has type $\alpha \longrightarrow \beta$ **and** x has type α.

Type Synthesis and Inference

- Type checking can be done by either synthesis or inference.
- A rule for type synthesis builds up the type of an expression from the type of its sub-expressions.
  - This requires that identifiers have declared their types before use.

Example

if f has type $s \longrightarrow t$ **and** x has type s, **then** $f(x)$ has type t.

- A rule for type inference infers the type of an expression from the way it is used.

Example (α and β are type variables)

if $f(x)$ is an expression, **then** there are α and β such that $f(x)$ has type $\alpha \longrightarrow \beta$ **and** x has type α.

Type Synthesis and Inference

- Type checking can be done by either synthesis or inference.
- A rule for type synthesis builds up the type of an expression from the type of its sub-expressions.
  - This requires that identifiers have declared their types before use.

Example

if f has type $s \longrightarrow t$ **and** x has type s, **then** $f(x)$ has type t.

- A rule for type inference infers the type of an expression from the way it is used.

Example (α and β are type variables)

if $f(x)$ is an expression, **then** there are α and β such that $f(x)$ has type $\alpha \longrightarrow \beta$ **and** x has type α.

Type Synthesis and Inference

- Type checking can be done by either synthesis or inference.
- A rule for type synthesis builds up the type of an expression from the type of its sub-expressions.
  - This requires that identifiers have declared their types before use.

Example

if f has type $s \longrightarrow t$ **and** x has type s, **then** $f(x)$ has type t.

- A rule for type inference infers the type of an expression from the way it is used.

Example (α and β are type variables)

if $f(x)$ is an expression, **then** there are α and β such that $f(x)$ has type $\alpha \longrightarrow \beta$ **and** x has type α.

Type Synthesis and Inference

- Type checking can be done by either synthesis or inference.
- A rule for type synthesis builds up the type of an expression from the type of its sub-expressions.
    - This requires that identifiers have declared their types before use.

Example

if f has type $s \longrightarrow t$ **and** x has type s, **then** $f(x)$ has type t.

- A rule for type inference infers the type of an expression from the way it is used.

Example (α and β are type variables)

if $f(x)$ is an expression, **then** there are α and β such that $f(x)$ has type $\alpha \longrightarrow \beta$ **and** x has type α.

Type Conversions

- Some programming languages allow changing the type of an expression.
- Valid changes are governed by rules of the type system.
- Conversions done implicitly by the compiler are referred to as coercions.
- Conversions explicitly forced by the user are referred to as casts.
- Coercions typically take place by converting from a type to a *wider* type; such conversions are information-preserving.
- Casts can also convert from a type to a *narrower* type; such conversions may compromise precision.

Type Conversions

- Some programming languages allow changing the type of an expression.
- Valid changes are governed by rules of the type system.
- Conversions done implicitly by the compiler are referred to as coercions.
- Conversions explicitly forced by the user are referred to as casts.
- Coercions typically take place by converting from a type to a *wider* type; such conversions are information-preserving.
- Casts can also convert from a type to a *narrower* type; such conversions may compromise precision.

Type Conversions

- Some programming languages allow changing the type of an expression.
- Valid changes are governed by rules of the type system.
- Conversions done implicitly by the compiler are referred to as coercions.
- Conversions explicitly forced by the user are referred to as casts.
- Coercions typically take place by converting from a type to a *wider* type; such conversions are information-preserving.
- Casts can also convert from a type to a *narrower* type; such conversions may compromise precision.

Type Conversions

- Some programming languages allow changing the type of an expression.
- Valid changes are governed by rules of the type system.
- Conversions done implicitly by the compiler are referred to as coercions.
- Conversions explicitly forced by the user are referred to as casts.
- Coercions typically take place by converting from a type to a *wider* type; such conversions are information-preserving.
- Casts can also convert from a type to a *narrower* type; such conversions may compromise precision.

Type Conversions

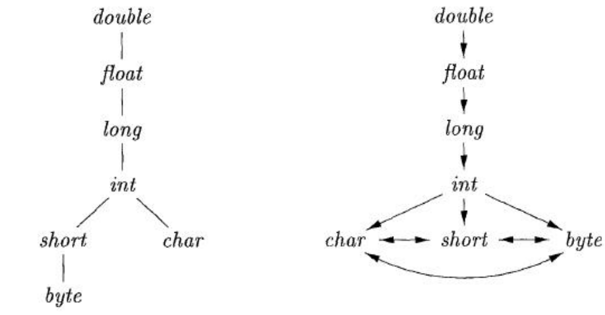
- Some programming languages allow changing the type of an expression.
- Valid changes are governed by rules of the type system.
- Conversions done implicitly by the compiler are referred to as coercions.
- Conversions explicitly forced by the user are referred to as casts.
- Coercions typically take place by converting from a type to a *wider* type; such conversions are information-preserving.
- Casts can also convert from a type to a *narrower* type; such conversions may compromise precision.

Type Conversions

- Some programming languages allow changing the type of an expression.
- Valid changes are governed by rules of the type system.
- Conversions done implicitly by the compiler are referred to as coercions.
- Conversions explicitly forced by the user are referred to as casts.
- Coercions typically take place by converting from a type to a *wider* type; such conversions are information-preserving.
- Casts can also convert from a type to a *narrower* type; such conversions may compromise precision.

Java Conversion Rules

Example



(a) Widening conversions (b) Narrowing conversions

Coercion

- Coercion uses two important functions.
  1. $max(t_1, t_2)$ returns the least-upper bound of t_1 and t_2 in the widening hierarchy.
  2. $widen(a, t, w)$ returns an address resulting from widening address a from type t to type w. If $t = w$, a is returned, else a new address is generated.

Coercion

- Coercion uses two important functions.
  1. $max(t_1, t_2)$ returns the least-upper bound of t_1 and t_2 in the widening hierarchy.
  2. $widen(a, t, w)$ returns an address resulting from widening address a from type t to type w. If $t = w$, a is returned, else a new address is generated.

Pseudo-Code for Widening

```
Addr widen(Addr a, Type t, Type w)
    if ( t = w ) return a;
    else if ( t = integer and w = float ) {
        temp = new Temp();
        gen(temp '=' '(float)' a);
        return temp;
    }
    else error;
}
```

Addition

Example

$$
\begin{aligned}
E \quad \longrightarrow \quad E_1 + E_2 \quad & \{E.type = max(E_1.type, E_2.type) \\
& a_1 = widen(E_1.addr, E_1.type, E.type) \\
& a_2 = widen(E_2.addr, E_2.type, E.type) \\
& E.addr = \textbf{new}\ temp() \\
& gen(E.addr\ '='\ a_1\ '+'\ a_2\}
\end{aligned}
$$

Overloaded Functions and Operators

- Type conversion, together with appropriate rules, may be used to accommodate function and operator overloading.
- The following rule resolves overloading when the type of an expression is determined by the types of operands.

$$\textbf{if } f \text{ can have type } s_i \longrightarrow t_i, \text{ for } 1 \leq i \leq n, \text{ where } s_i \neq s_j \text{ for } i \neq j$$
$$\textbf{and } x \text{ is of type } s_k, \text{ for some } 1 \leq k \leq n$$
$$\textbf{then } f(x) \text{ is of type } t_k$$

Overloaded Functions and Operators

- Type conversion, together with appropriate rules, may be used to accommodate function and operator overloading.
- The following rule resolves overloading when the type of an expression is determined by the types of operands.

$$\begin{array}{c}\textbf{if } f \text{ can have type } s_i \longrightarrow t_i, \text{ for } 1 \leq i \leq n, \text{ where } s_i \neq s_j \text{ for } i \neq j \\ \textbf{and } x \text{ is of type } s_k, \text{ for some } 1 \leq k \leq n \\ \textbf{then } f(x) \text{ is of type } t_k\end{array}$$

Overloaded Functions and Operators

- Type conversion, together with appropriate rules, may be used to accommodate function and operator overloading.
- The following rule resolves overloading when the type of an expression is determined by the types of operands.

if f can have type $s_i \longrightarrow t_i$, for $1 \leq i \leq n$, where $s_i \neq s_j$ for $i \neq j$
and x is of type s_k, for some $1 \leq k \leq n$
then $f(x)$ is of type t_k

Polymorphic Functions

- A function is polymorphic if it can be executed with arguments of different types.
- Note that this is not the same as overloading.
- Polymorphic functions often have type variables in their type expressions.
- The presence of such functions complicate type checking.

Polymorphic Functions

- A function is polymorphic if it can be executed with arguments of different types.
- Note that this is not the same as overloading.
- Polymorphic functions often have type variables in their type expressions.
- The presence of such functions complicate type checking.

Polymorphic Functions

- A function is polymorphic if it can be executed with arguments of different types.
- Note that this is not the same as overloading.
- Polymorphic functions often have type variables in their type expressions.
- The presence of such functions complicate type checking.

Polymorphic Functions

- A function is polymorphic if it can be executed with arguments of different types.
- Note that this is not the same as overloading.
- Polymorphic functions often have type variables in their type expressions.
- The presence of such functions complicate type checking.

ML Lists

Example

```
fun length(x) =
  if null(x) then 0 else length(tl(x)) + 1
```

What is the type of `length`?

Type Inference Algorithm

- For function definitions $id1(id2) = E$.
  1. Let $id1 : \alpha \longrightarrow \beta$.
  2. Let $id2 : \alpha$.
  3. Infer a type for E.
  4. Bind α and β accordingly.
- For function calls $E1(E2)$.
  1. Infer a type for $E1$—$s \longrightarrow t$, for example.
  2. Infer a type for $E2$—s', for example.
  3. Unify s and s'—suppose the result is μ.
  4. If $\mu = fail$, then signal an error, else the inferred type of $E1(E2)$ is $\mu(t)$.

Type Inference Algorithm

- For function definitions $id1(id2) = E$.
  1. Let $id1 : \alpha \longrightarrow \beta$.
  2. Let $id2 : \alpha$.
  3. Infer a type for E.
  4. Bind α and β accordingly.
- For function calls $E1(E2)$.
  1. Infer a type for $E1$—$s \longrightarrow t$, for example.
  2. Infer a type for $E2$—s', for example.
  3. Unify s and s'—suppose the result is μ.
  4. If $\mu = fail$, then signal an error, else the inferred type of $E1(E2)$ is $\mu(t)$.

Type Inference Algorithm

- For function definitions $id1(id2) = E$.
  1. Let $id1 : \alpha \longrightarrow \beta$.
  2. Let $id2 : \alpha$.
  3. Infer a type for E.
  4. Bind α and β accordingly.
- For function calls $E1(E2)$.
  1. Infer a type for $E1$—$s \longrightarrow t$, for example.
  2. Infer a type for $E2$—s', for example.
  3. Unify s and s'—suppose the result is μ.
  4. If $\mu = fail$, then signal an error, else the inferred type of $E1(E2)$ is $\mu(t)$.

Type Inference Algorithm

- For function definitions $id1(id2) = E$.
  1. Let $id1 : \alpha \longrightarrow \beta$.
  2. Let $id2 : \alpha$.
  3. Infer a type for E.
  4. Bind α and β accordingly.
- For function calls $E1(E2)$.
  1. Infer a type for $E1$—$s \longrightarrow t$, for example.
  2. Infer a type for $E2$—s', for example.
  3. Unify s and s'—suppose the result is μ.
  4. If $\mu = fail$, then signal an error, else the inferred type of $E1(E2)$ is $\mu(t)$.

Type Inference Algorithm

- For function definitions $id1(id2) = E$.
  1. Let $id1 : \alpha \longrightarrow \beta$.
  2. Let $id2 : \alpha$.
  3. Infer a type for E.
  4. Bind α and β accordingly.
- For function calls $E1(E2)$.
  1. Infer a type for $E1$—$s \longrightarrow t$, for example.
  2. Infer a type for $E2$—s', for example.
  3. Unify s and s'—suppose the result is μ.
  4. If $\mu = fail$, then signal an error, else the inferred type of $E1(E2)$ is $\mu(t)$.

Type Inference Algorithm

- For function definitions $id1(id2) = E$.
  1. Let $id1 : \alpha \longrightarrow \beta$.
  2. Let $id2 : \alpha$.
  3. Infer a type for E.
  4. Bind α and β accordingly.
- For function calls $E1(E2)$.
  1. Infer a type for $E1$—$s \longrightarrow t$, for example.
  2. Infer a type for $E2$—s', for example.
  3. Unify s and s'—suppose the result is μ.
  4. If $\mu = fail$, then signal an error, else the inferred type of $E1(E2)$ is $\mu(t)$.

Type Inference Algorithm

- For function definitions $id1(id2) = E$.
  1. Let $id1 : \alpha \longrightarrow \beta$.
  2. Let $id2 : \alpha$.
  3. Infer a type for E.
  4. Bind α and β accordingly.
- For function calls $E1(E2)$.
  1. Infer a type for $E1$—$s \longrightarrow t$, for example.
  2. Infer a type for $E2$—s', for example.
  3. Unify s and s'—suppose the result is μ.
  4. If $\mu = fail$, then signal an error, else the inferred type of $E1(E2)$ is $\mu(t)$.

Type Inference Algorithm

- For function definitions $id1(id2) = E$.
  1. Let $id1 : \alpha \longrightarrow \beta$.
  2. Let $id2 : \alpha$.
  3. Infer a type for E.
  4. Bind α and β accordingly.
- For function calls $E1(E2)$.
  1. Infer a type for $E1$—$s \longrightarrow t$, for example.
  2. Infer a type for $E2$—s', for example.
  3. Unify s and s'—suppose the result is μ.
  4. If $\mu = fail$, then signal an error, else the inferred type of $E1(E2)$ is $\mu(t)$.

Type Inference Algorithm

- For function definitions $id1(id2) = E$.
  1. Let $id1 : \alpha \longrightarrow \beta$.
  2. Let $id2 : \alpha$.
  3. Infer a type for E.
  4. Bind α and β accordingly.
- For function calls $E1(E2)$.
  1. Infer a type for $E1$—$s \longrightarrow t$, for example.
  2. Infer a type for $E2$—s', for example.
  3. Unify s and s'—suppose the result is μ.
  4. If $\mu = fail$, then signal an error, else the inferred type of $E1(E2)$ is $\mu(t)$.

Type Inference Algorithm

- For function definitions $id1(id2) = E$.
  1. Let $id1 : \alpha \longrightarrow \beta$.
  2. Let $id2 : \alpha$.
  3. Infer a type for E.
  4. Bind α and β accordingly.
- For function calls $E1(E2)$.
  1. Infer a type for $E1$—$s \longrightarrow t$, for example.
  2. Infer a type for $E2$—s', for example.
  3. Unify s and s'—suppose the result is μ.
  4. If $\mu = fail$, then signal an error, else the inferred type of $E1(E2)$ is $\mu(t)$.

ML Lists, Again

LINE	EXPRESSION : TYPE	UNIFY
1)	$length : \beta \to \gamma$	
2)	$x : \beta$	
3)	$\textbf{if} : boolean \times \alpha_i \times \alpha_i \to \alpha_i$	
4)	$null : list(\alpha_n) \to boolean$	
5)	$null(x) : boolean$	$list(\alpha_n) = \beta$
6)	$0 : integer$	$\alpha_i = integer$
7)	$+ : integer \times integer \to integer$	
8)	$tl : list(\alpha_t) \to list(\alpha_t)$	
9)	$tl(x) : list(\alpha_t)$	$list(\alpha_t) = list(\alpha_n)$
10)	$length(tl(x)) : \gamma$	$\gamma = integer$
11)	$1 : integer$	
12)	$length(tl(x)) + 1 : integer$	
13)	$\textbf{if}(\ \cdots\ ) : integer$	