

Programming Languages, CPSC 604—Slides 5

Simple Types

Jaakko Järvi

February 3, 2009

Outline

- 1 Types for \mathcal{NB}
- 2 Properties of \mathcal{NB} typing

Problems with the “untyped” evaluator

- Our evaluator for the language \mathcal{NB} works just fine...

Problems with the “untyped” evaluator

- Our evaluator for the language \mathcal{NB} works just fine...
- If we use it nicely

Problems with the “untyped” evaluator

- Our evaluator for the language \mathcal{NB} works just fine...
- If we use it nicely
- If not, terms such as:

```
iszero true  
if 0 then 0 else 0
```

lead to the evaluator getting “**stuck**”

- There are terms that are in normal form, but are not values!
- Terms that evaluate to stuck states represent invalid/meaningless programs
- Fix: Reject such terms
- Note: we are interested in rejecting terms that are accepted by the CF-grammar, but evaluation gets stuck

Strategies for dealing with meaningless programs

- “C way”
- Reject most meaningless programs: `char* p = 1;`
- but allow some:

```
union {  
    char* p;  
    int i;  
} my_union;
```

```
void foo() {  
    my_union.i = 1;  
    char* p = my_union.p;  
    *p = 'a';  
}
```

Strategies for dealing with meaningless programs

- “C way”
- Reject most meaningless programs: `char* p = 1;`
- but allow some:

```
union {  
    char* p;  
    int i;  
} my_union;
```

```
void foo() {  
    my_union.i = 1;  
    char* p = my_union.p;  
    *p = 'a';  
}
```

- and just let the machine run and do whatever

Java way

- Reject some meaningless programs at compile-time:
`Int i = "Erroneous";`
- Reject more at run-time (or make error states part of well-defined semantics)

```
interface Stack
{
    void push(Object elem);
    Object pop();
}

class MyStack { ... }

Stack s = new MyStack();
s.push(1);
s.push(2);
s.push("supercalifraci...");

Int s = (Int) s.pop(); // throws an exception
```

Scheme way

- Reject none at compile-time
- All (well, most) at run-time

`(car (cons 1 2))` ; *ok*

`(car 5)` ; *error at run-time*

Scheme way

- Reject none at compile-time
- All (well, most) at run-time

`(car (cons 1 2))` ; *ok*

`(car 5)` ; *error at run-time*

- Captures exactly (not quite) the set of safe programs
- Moving between data and code is trivial

`(quote (+ 1 2 3))` ; *this is just data*

`(eval (quote (+ 1 2 3)) (scheme-report-environment 5))`

Ideal(?) way

- Reject all meaningless programs at compile-time
- Is this possible?

Ideal(?) way

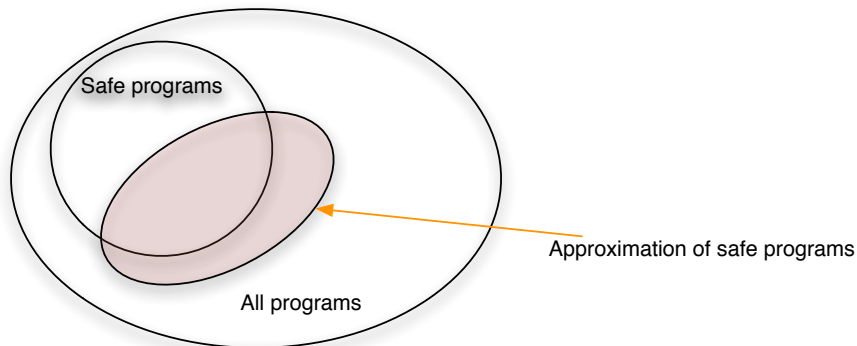
- Reject all meaningless programs at compile-time
- Is this possible? Yes, easily.

Ideal(?) way

- Reject all meaningless programs at compile-time
- Is this possible? Yes, easily.
- How about: “Reject all meaningless programs at compile-time and no meaningful ones?”

Ideal(?) way

- Reject all meaningless programs at compile-time
- Is this possible? Yes, easily.
- How about: “Reject all meaningless programs at compile-time and no meaningful ones?”



Ideal(?) way

- In practical languages not possible; type-checking can reject more programs than the grammar does, but not all
- In general an undecidable problem (think e.g. of detecting “division-by-zero” errors)
- Note that exact type checking most likely is not desirable. Why?

Ideal(?) way

- In practical languages not possible; type-checking can reject more programs than the grammar does, but not all
- In general an undecidable problem (think e.g. of detecting “division-by-zero” errors)
- Note that exact type checking most likely is not desirable. Why?
- E.g. prevent breaking abstractions, reject brittle code, ...

Quote

The fundamental problem addressed by a type theory is to insure that programs have meaning. The fundamental problem caused by a type theory is that meaningful programs may not have meanings ascribed to them. The quest for richer type systems results from this tension. [Mark Manasse]

History (a tiny bit)

- Type theory deals with classifying things into “types”
- Originated as a response to Russell’s paradox:
Consider the set of all sets that are not members of themselves. This set is a member of itself if and only if it is not a member of itself!
- We can operate on a much more practical level. On first approximation, types are sets of terms/objects/values in a programming language.
- Pierce:
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.

Types in \mathcal{NB}

- \mathcal{NB} has two categories/classes/sets/... of terms
 - truth values
 - numeric values
- Now we classify `true` and `false` to be of type `Bool`
- and numeric values of type `Nat`
- Informally, by saying “term t is of type T ”, we imply that that we can see (without evaluating t) that t evaluates to some normal form t' , which has type T .
- Terminology: all of the following mean the same
 - t is of type T
 - t has type T
 - t belongs to type T
 - type of t is T

Typing relation

- The notation for t is of type T is:

$$t : T$$

or

$$t \in T$$

- And more commonly:

$$\Gamma \vdash t : T$$

where Γ is the **context**, or **typing environment**

- For \mathcal{NB} , elements of the typing relation are pairs, no typing environment is necessary
- Typing relation is commonly defined with inference rules

Reminder: Language \mathcal{NB}

t	::=		terms:
		v	value
		$\text{if } t_1 \text{ then } t_2 \text{ else } t_3$	conditional
		$\text{succ } t$	successor
		$\text{pred } t$	predecessor
		$\text{iszero } t$	test for zero
v	::=		values:
		true	constant true
		false	constant false
		nv	numeric value
nv	::=		numeric values:
		0	zero value
		$\text{succ } nv$	successor value

Reminder \mathcal{NB} evaluation rules

$$\text{E-Iszero} \quad \frac{t \rightarrow t'}{\text{iszero } t \rightarrow \text{iszero } t'}$$

$$\text{E-IszeroZero} \quad \text{iszero } 0 \rightarrow \text{true}$$

$$\text{E-IszeroSucc} \quad \text{iszero } (\text{succ } nv) \rightarrow \text{false}$$

$$\text{E-Succ} \quad \frac{t \rightarrow t'}{\text{succ } t \rightarrow \text{succ } t'}$$

$$\text{E-Pred} \quad \frac{t \rightarrow t'}{\text{pred } t \rightarrow \text{pred } t'}$$

$$\text{E-PredZero} \quad \text{pred } 0 \rightarrow 0$$

$$\text{E-PredSucc} \quad \text{pred } (\text{succ } nv) \rightarrow nv$$

$$\text{E-IfTrue} \quad \text{if true then } t_2 \text{ else } t_3 \rightarrow t_2$$

$$\text{E-IfFalse} \quad \text{if false then } t_2 \text{ else } t_3 \rightarrow t_3$$

$$\text{E-If} \quad \frac{t_1 \rightarrow t'_1}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rightarrow \text{if } t'_1 \text{ then } t_2 \text{ else } t_3}$$

\mathcal{NB} typing rules

- First, we need a few new syntactic forms

$T ::=$
Bool
the Boolean type
Nat
the type of numeric values

- And typing rules:

T-True
 $\text{true} : \text{Bool}$

T-False
 $\text{false} : \text{Bool}$

T-If

$$\frac{t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T}$$

T-Zero
 $0 : \text{Nat}$

T-Succ

$$\frac{t : \text{Nat}}{\text{succ } t : \text{Nat}}$$

T-Pred

$$\frac{t : \text{Nat}}{\text{pred } t : \text{Nat}}$$

T-Iszero

$$\frac{t : \text{Nat}}{\text{iszero } t : \text{Bool}}$$

\mathcal{NB} typing relation

Definition

The typing relation for \mathcal{NB} is the smallest binary relation between terms and types satisfying all instances of the inference rules on the previous slide.

We say that a term t is **typable**, or **well-typed** if there is some T such that $t : T$.

Examples

- What are the types of these terms?
 - `succ (succ 0)`
 - `if iszero 0 then 0 else succ 0`
 - `if iszero 0 then 0 else false`
- Draw the derivation trees

Derivations

- $\text{succ}(\text{succ } 0)$

$$\frac{0 : \text{Nat}}{\text{succ } 0 : \text{Nat}} \text{ T-Succ}$$

$$\frac{\text{succ } 0 : \text{Nat}}{\text{succ } (\text{succ } 0) : \text{Nat}} \text{ T-Succ}$$

- $\text{if iszero } 0 \text{ then } 0 \text{ else succ } 0$

$$\frac{\frac{0 : \text{Nat T-Zero}}{\text{iszero } 0 : \text{Bool}} \text{ T-Succ} \quad 0 : \text{Nat T-Zero} \quad \frac{0 : \text{Nat T-Zero}}{\text{succ } 0 : \text{Nat}} \text{ T-Succ}}{\text{if iszero } 0 \text{ then } 0 \text{ else succ } 0 : \text{Nat}} \text{ T-If}$$

- $\text{if iszero } 0 \text{ then } 0 \text{ else false}$ Not well-typed!

$$\frac{\frac{0 : \text{Nat T-Zero}}{\text{iszero } 0 : \text{Bool}} \text{ T-Succ} \quad 0 : T(?) \quad \text{false} : T(?)}{\text{if iszero } 0 \text{ then } 0 \text{ else false} : \text{Nat}} \text{ T-If}$$

Outline

- 1 Types for \mathcal{NB}
- 2 Properties of \mathcal{NB} typing

Type safety

- We hope that with the type system we defined, the language possesses the property of **type safety** (also **soundness**)
- This means that if t is a well-typed term, it evaluates to a value rather than getting “stuck”
 - Reminder: A **stuck** state is a term for which no evaluation rule applies, but the term is not a value.
- Proving soundness is typically established by proving
 - **Progress**: a well typed term is either a value, or some evaluation rule applies
 - **Preservation**: evaluation relation preserves well-typedness of a term
- We first establish a few useful lemmas...

Inversion

- The **Inversion lemma** reads the typing relation backwards, allowing us to limit the possible types for many terms (by looking at their top-level syntactic form)

Lemma (Inversion of typing relation)

- 1 If $\text{true} : R$, then $R = \text{Bool}$
- 2 If $\text{false} : R$, then $R = \text{Bool}$
- 3 If $\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : R$, then $t_1 : \text{Bool}$, $t_2 : R$, and $t_3 : R$.
- 4 If $0 : R$, then $R = \text{Nat}$
- 5 If $\text{succ } t_1 : R$, then $R = \text{Nat}$ and $t_1 : \text{Nat}$
- 6 If $\text{pred } t_1 : R$, then $R = \text{Nat}$ and $t_1 : \text{Nat}$
- 7 If $\text{iszero } t_1 : R$, then $R = \text{Bool}$ and $t_1 : \text{Nat}$

Proof.

Follows directly from the typing relation.

Uniqueness of types

Theorem (Uniqueness of types)

No term has more than one type. That is, if $t : T_1$ and $t : T_2$, then $T_1 = T_2$.

Proof.

By induction on the structure of t (using inversion lemma). □

- In fact, a stronger property holds for \mathcal{NB} :

Theorem (Uniqueness of typing derivations)

If $t : T_1$ and $t : T_2$, then the typing derivations of $t : T_1$ and $t : T_2$ are equal.

About uniqueness

- Uniqueness theorem does not hold for more complex languages
- Can you find examples of such type systems?

About uniqueness

- Uniqueness theorem does not hold for more complex languages
- Can you find examples of such type systems?
- For example systems with **subtyping**

```
class A {};  
class B : public A {};
```

```
B b; // b has both type B and type A
```

Canonical forms

- This lemma allows us to limit the shapes of terms of different types

Lemma (Canonical forms)

- ① If v is a value and has type `Bool`, then v is either `true` or `false`.
- ② If v is a value and has type `Nat`, then v is a *numeric value* as specified in our grammar.

Proof.

Immediate from the grammar and inversion lemma. □

Progress

- This is the first half of our main theorem

Theorem (Progress)

Assume $t : T$ (i.e., t is well-typed). Then, either t is a value, or $t \rightarrow t'$ for some t' .

Proof.

By induction on typing derivation $t : T$. Trivial if the last rule used is T-True, T-False, or T-Zero (t is a value).

Case T-If: t is of the form `if t_1 then t_2 else t_3` , where $t_1 : \text{Bool}$, $t_2 : T$, and $t_3 : T$. By the induction hypothesis, t_1 , t_2 , and t_3 each either are values or evaluate (respectively) to some terms t'_1 , t'_2 , and t'_3 . If t_1 is a value, from the canonical forms lemma we see it must be either `true` or `false`, and thus either $t \rightarrow t_2$ or $t \rightarrow t_3$ using E-IfTrue or E-IfFalse. If $t_1 \rightarrow t'_1$, then $t \rightarrow \text{if } t'_1 \text{ then } t_2 \text{ else } t_3$ by E-If.

... Progress

Theorem (Progress)

Assume $t : T$ (i.e., t is well-typed). Then, either t is a value, or $t \rightarrow t'$ for some t' .

Proof.

Case T-Pred: t is of the form $\text{pred } t_1$, where $t_1 : \text{Nat}$. By the induction hypothesis, t_1 is either a value or evaluates to some term t'_1 . If t_1 is a value, from the canonical forms lemma we see it must be a numeric value, and thus either $t_1 = 0$ or $t_1 = \text{succ } nv$. If $t_1 = 0$, then $t = \text{pred } 0 \rightarrow 0$ using the rule E-PredZero. If $t_1 = \text{succ } nv$, then $t = \text{pred } (\text{succ } nv) \rightarrow nv$. If $t_1 \rightarrow t'_1$, then rule E-Pred applies and $t = \text{pred } t_1 \rightarrow \text{pred } t'_1$.

Case T-Succ: Exercise.



The second part: Preservation

Theorem (Preservation of well-typedness)

If $t : T$ and $t \rightarrow t'$, then $t' : T'$, for some T' .

- For \mathcal{NB} , we can prove a stronger preservation theorem:

Theorem (Preservation of typing)

If $t : T$ and $t \rightarrow t'$, then $t' : T$.

- Preservation theorem is also known as **subject reduction**

... Preservation

Theorem (Preservation of typing)

If $t : T$ and $t \rightarrow t'$, then $t' : T$.

Proof.

By induction on typing derivation $t : T$.

Vacuously true for T-True, T-False, and T-Zero.

Case T-If: t is of the form $\text{if } t_1 \text{ then } t_2 \text{ else } t_3$, where $t_1 : \text{Bool}$, $t_2 : T$, and $t_3 : T$. There are three possible rules for $t \rightarrow t'$:

- ① If $t_1 = \text{true}$, by E-IfTrue t evaluates to t_2 which is of type T .
- ② If $t_1 = \text{false}$, by E-IfFalse t evaluates to t_3 which has type T .
- ③ Otherwise E-If must apply and $t_1 \rightarrow t'_1$ for some t'_1 . By induction hypothesis, t'_1 is of the same type as t_1 : type Bool . Thus $t' = \text{if } t'_1 \text{ then } t_2 \text{ else } t_3$, where $t'_1 : \text{Bool}$, $t_2 : T$, and $t_3 : T$. The type of this t' is thus T .

Cases T-Pred and T-Succ: Exercise.