Untyped Arithmetic Expressions

This chapter lays the groundwork for understanding type systems by formally addressing the syntax and semantics of simple programming languages. It introduces key concepts like **abstract syntax**, **inductive definitions**, **evaluation**, and **run-time errors**. The discussion starts with a basic language of numbers and booleans, then progresses to the untyped lambda-calculus.

Introduction

The language in this chapter includes:

- Boolean constants: **true** and **false**
- Conditional expressions: **if-then-else**
- Numeric constant: **0**
- Arithmetic operators: **succ** $successor$ and **pred** $predecessor$
- Testing operation: **iszero** $returnstrueifappliedto0, falseotherwise$

These forms are summarized by the following grammar:

```
t ::=
    true              constant true
    false             constant false
    if t then t else t  conditional
    0                 constant zero
    succ t            successor
    pred t            predecessor
    iszero t          zero test
```

In this grammar, `t` is a **metavariable**, a placeholder for terms in the language, not a variable within the language itself $whichwillbeintroducedlater$.

The terms "term" and "expression" are used interchangeably in this chapter. Later, "expression" will encompass terms, types, and kinds, while "term" will refer specifically to phrases representing computations.

Examples of programs in this language and their evaluation results:

```
if false then 0 else 1; 1
iszero (pred (succ 0)); true
```

The symbol `⟿` indicates the result of evaluating an expression. Results are either boolean constants or numbers (nested applications of `succ` to `0`). These are called **values**.

Note that the syntax allows for potentially problematic terms like `succ true` and `if 0 then 0 else 0`. These represent the kinds of nonsensical programs that a type system aims to prevent.

Syntax

The syntax of the language can be defined in several equivalent ways. One way is the grammar described above. Another is through an inductive definition.

Inductive Definition of Terms

> The set of terms is the smallest set T such that:
>
> 1. `{true, false, 0} ⊆ T`
> 2. If `t1 ∈ T`, then `{succ t1, pred t1, iszero t1} ⊆ T`
> 3. If `t1 ∈ T`, `t2 ∈ T`, and `t3 ∈ T`, then `if t1 then t2 else t3 ∈ T`

This definition defines T as a set of trees, not strings, with parentheses used to clarify the structure. Another shorthand for the inductive definition of terms uses inference rules.

Terms by Inference Rules

> The set of terms is defined by the following rules:

$$\frac{}{true \in T} \quad \frac{}{false \in T} \quad \frac{}{0 \in T}$$

$$\frac{t_1 \in T}{succ\ t_1 \in T} \quad \frac{t_1 \in T}{pred\ t_1 \in T} \quad \frac{t_1 \in T}{iszero\ t_1 \in T}$$

$$\frac{t_1 \in T \quad t_2 \in T \quad t_3 \in T}{if\ t_1\ then\ t_2\ else\ t_3 \in T}$$

Each rule states that if the premises above the line are true, then the conclusion below the line can be derived. Rules without premises are called **axioms**.

Another definition style involves an explicit procedure for generating the elements of T.

Concrete Definition of Terms

> For each natural number i, define a set Si as follows:
>
> - S0 = ∅
> - Si+1 = `{true, false, 0} ∪ {succ t1, pred t1, iszero t1 | t1 ∈ Si} ∪ {if t1 then t2 else t3 | t1, t2, t3 ∈ Si}`
>
> Finally, let $S = \bigcup_i S_i$.

This builds the set of terms iteratively, starting with constants and adding more complex terms in each step.

Equivalence of Definitions

The different definitions characterize the same set of terms. Proposition 3.2.6 states that T = S, which is proven by showing that S satisfies the conditions defining T and that any set satisfying those conditions contains S as a subset.

Induction on Terms

The characterization of the set of terms T allows for inductive definitions of functions over terms and inductive proofs of properties of terms.

Inductive Definitions

Example: The set of constants appearing in a term t, `Consts(t)`, is defined as:

```
Consts(true) = {true}
Consts(false) = {false}
Consts(0) = {0}
Consts(succ t1) = Consts(t1)
Consts(pred t1) = Consts(t1)
Consts(iszero t1) = Consts(t1)
Consts(if t1 then t2 else t3) = Consts(t1) ∪ Consts(t2) ∪ Consts(t3)
```

Another example: The size of a term t, `size(t)`, is defined as:

```
size(true) = 1
size(false) = 1
size(0) = 1
size(succ t1) = size(t1) + 1
size(pred t1) = size(t1) + 1
size(iszero t1) = size(t1) + 1
size(if t1 then t2 else t3) = size(t1) + size(t2) + size(t3) + 1
```

Similarly, the depth of a term t, `depth(t)`, is defined as:

```
depth(true) = 1
depth(false) = 1
depth(0) = 1
depth(succ t1) = depth(t1) + 1
depth(pred t1) = depth(t1) + 1
depth(iszero t1) = depth(t1) + 1
depth(if t1 then t2 else t3) = max(depth(t1), depth(t2), depth(t3)) + 1
```

Inductive Proofs

Lemma 3.3.3: The number of distinct constants in a term t is no greater than the size of t (`|Consts(t)| ≤ size(t)`). The proof proceeds by induction on the depth of t, considering each possible form of t $constant, successor, predecessor, iszero, orconditional$ and applying the induction hypothesis to the subterms.

Theorem 3.3.4 presents principles of induction on terms:

- **Induction on depth**: If `P(r)` holds for all `r` with `depth(r) < depth(s)`, and this implies `P(s)`, then `P(s)` holds for all `s`.
- **Induction on size**: If `P(r)` holds for all `r` with `size(r) < size(s)`, and this implies `P(s)`, then `P(s)` holds for all `s`.
- **Structural induction**: If `P(r)` holds for all immediate subterms `r` of `s`, and this implies `P(s)`, then `P(s)` holds for all `s`.

Semantic Styles

Formalizing the semantics of a language involves defining how terms are evaluated. There are three primary approaches:

1. **Operational semantics**: Specifies the behavior of a programming language by defining a simple abstract machine for it.
2. **Denotational semantics**: Takes a more abstract view of meaning, interpreting terms as mathematical objects.
3. **Axiomatic semantics**: Defines the meaning of a term by the laws that can be proven about it.

Operational semantics is the method used in this book.

Evaluation

Evaluation of boolean expressions.

Syntax and Values

The syntax defines the structure of terms (`t`) and a subset of terms called values (`v`), which are the final results of evaluation.

```
t ::=
    true
    false
    if t then t else t

v ::=
    true
    false
```

Evaluation Relation

The evaluation relation, written `t → t'`, means "t evaluates to t' in one step."

The evaluation rules are:

- **E-IfTrue**: If the guard is `true`, the conditional evaluates to the `then` part (`t2`).

  $$if\ true\ then\ t_2\ else\ t_3 \rightarrow t_2$$

- **E-IfFalse**: If the guard is `false`, the conditional evaluates to the `else` part (`t3`).

  $$if\ false\ then\ t_2\ else\ t_3 \rightarrow t_3$$

- **E-If**: If the guard `t1` evaluates to `t1'`, the whole conditional evaluates to `if t1' then t2 else t3`.

$$\frac{t_1 \rightarrow t_1'}{if\ t_1\ then\ t_2\ else\ t_3 \rightarrow if\ t_1'\ then\ t_2\ else\ t_3}$$

Here's a visual representation:

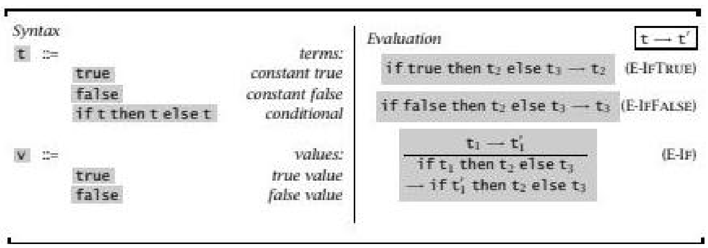

Figure 3-1: Booleans B. This image encapsulates the formal definitions of syntax and evaluation for boolean expressions, including the rules for conditional $if-then-else$ evaluation.

Evaluation Strategies for Conditionals

The evaluation of conditionals follows a specific strategy. Consider the term:

`if true then (if false then false else false) else true`

According to the **E-If** rule, we must evaluate the outer conditional first. This strategy mirrors the order of evaluation in common programming languages: evaluate the guard of a conditional before the conditional itself, recursively if the guard is also a conditional.

The **E-IfTrue** and **E-IfFalse** rules perform the actual evaluation, while **E-If** determines where to begin the evaluation. **E-IfTrue** and **E-IfFalse** are often called **computation rules**, and **E-If** is called a **congruence rule**.

> **Definition 3.5.1**: An instance of an inference rule is obtained by consistently replacing each metavariable by the same term in the rule's conclusion and all its premises $if any$.

For example:

`if true then true else (if false then false else false) → true`

is an instance of **E-IfTrue**, where `t2` is replaced by `true` and `t3` is replaced by `if false then false else false`.

> **Definition 3.5.2**: A rule is satisfied by a relation if, for each instance of the rule, either the conclusion is in the relation or one of the premises is not.

> **Definition 3.5.3**: The one-step evaluation relation → is the smallest binary relation on terms satisfying the three rules in Figure 3-1.

A statement `t → t` is **derivable** if it's justified by the rules, either as an instance of **E-IfTrue** or **E-IfFalse**, or as the conclusion of an instance of **E-If** with a derivable premise.

Consider the following abbreviations:

- `t = if true then false else false`
- `u = if s then true else true`
- `s = if false then true else true`

The derivability of the statement `$if t then false else false → if u then false else false$` is demonstrated by the following derivation tree:

$$E-IfTrue s \to false E-If t \to u E-If if t then false else false \to if u then false else false$$

This "tree" doesn't branch because each evaluation rule has at most one premise.

Properties of the Evaluation Relation

Derivation trees are useful when reasoning about the evaluation relation, leading to a proof technique called **induction on derivations**.

> **Theorem 3.5.4**
>
> $$Determinacy of one-step evaluation$$
>
> : If t → t' and t → t'', then t' = t''.

Proof: By induction on a derivation of `t → t'`.

Induction Steps:

- If the last rule used in the derivation of `t → t'` is **E-IfTrue**:
  - `t` has the form `if t1 then t2 else t3`, where `t1 = true`.
  - The last rule in the derivation of `t → t''` cannot be **E-IfFalse** (since `t1 = true`) or **E-If** (since `true` doesn't evaluate to anything).
  - Thus, the last rule in the second derivation is **E-IfTrue**, implying `t' = t''`.
- If the last rule used in the derivation of `t → t'` is **E-IfFalse**:
  - Similar logic applies, and the result is immediate.
- If the last rule used in the derivation of `t → t'` is **E-If**:
  - `t` has the form `if t1 then t2 else t3`, where `t1 → t1'` for some `t1'`.
  - The last rule in the derivation of `t → t''` can only be **E-If**, so `t''` has the form `if t1'' then t2 else t3` and `t1 → t1''` for some `t1''`.
  - The induction hypothesis applies, yielding `t1' = t1''`. Thus, `t' = t''`.

Normal Forms and Values

> **Definition 3.5.6**: A term `t` is in **normal form** if no evaluation rule applies to it—i.e., if there is no `t'` such that `t → t'`.

true and false are normal forms in the system.

> Theorem 3.5.7: Every value is in normal form.

In the current system, the converse is also true.

> Theorem 3.5.8: If `t` is in normal form, then `t` is a value.

Proof: By structural induction on `t`. If `t` is not a value, it must have the form `if t1 then t2 else t3`.

- If `t1 = true` or `t1 = false`, then `t` is not a normal form.
- If `t1` is neither `true` nor `false`, then it is not a value. By the induction hypothesis, `t1` is not a normal form. Thus, `t` is not a normal form.

Multi-Step Evaluation

> Definition 3.5.9: The multi-step evaluation relation →* is the reflexive, transitive closure of one-step evaluation.

This means:

1. If `t → t'`, then `t →* t'`.
2. `t →* t` for all `t`.
3. If `t →* t'` and `t' →* t''`, then `t →* t''`.

> Theorem 3.5.11
>
> $$Uniqueness of normal forms$$
>
> : If `t →* u` and `t →* u'`, where `u` and `u'` are both normal forms, then `u = u'`.

Termination of Evaluation

> Theorem 3.5.12
>
> $$Termination of Evaluation$$
>
> : For every term `t` there is some normal form `t'` such that `t →* t'`.

Each evaluation step reduces the size of the term, and the size is a termination measure because the usual order on the natural numbers is well founded.

Extending Evaluation to Arithmetic Expressions

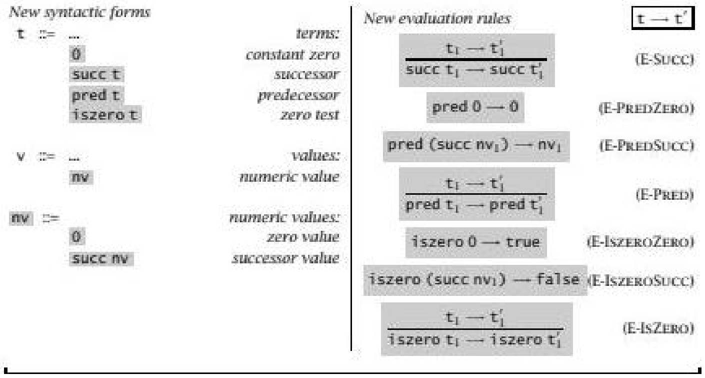

Figure 3-2: Arithmetic expressions NB

The image above introduces the syntax and evaluation rules for arithmetic expressions. The syntax includes terms for zero, successor, predecessor, and iszero, and it defines numeric values. The evaluation rules detail how these arithmetic operations behave, specifying, for example, how `pred` and `iszero` operate on numeric values.

The definition of values is extended to include numeric values, which are either 0 or the successor of another numeric value.

The evaluation rules include:

- Computation rules: E-PredZero, E-PredSucc, E-IszeroZero, and E-IszeroSucc. These rules define the behavior of `pred` and `iszero` when applied to numbers.
- Congruence rules: E-Succ, E-Pred, and E-Iszero. These rules direct the evaluation into the "first" subterm of a compound term.

For example, the unique next step in the evaluation of `pred (succ (pred 0))` has the following derivation tree:

$$E - PredZeropred0 \rightarrow 0E - Succsucc(pred0) \rightarrow succ0E - Predpred(succ(pred0)) \rightarrow pred(succ0)$$

Stuck Terms and Run-time Errors

Terms like `pred 0` and `succ false` need to be handled. Under the given rules, `pred 0` evaluates to 0.

> Definition 3.5.15: A closed term is stuck if it is in normal form but not a value.

Stuckness represents a simple notion of run-time error, indicating that the operational semantics doesn't know what to do because the program has reached a "meaningless state."