Theorem 1. *$H(P,x)$ is undecidable.*

Proof. Assume program $Q(P,x)$ computes H (somehow). Construct another program $D(P)$ such that

$D(P)$:
  if $Q(P,P)$ = "halts" then loop
  else halt

In other words, $D(P)$ exhibits the opposite halting behavior of $P(P)$.

Now, consider the effect of executing $D(D)$. According to the program definition, $D(D)$ must halt if $D(D)$ would run forever, *and* $D(D)$ must run forever if $D(D)$ would halt. Because $D(D)$ cannot both halt and run forever, this is a contradiction. Therefore the assumption that Q computes H is false. We made no further assumption beyond H being decidable, therefore H must be undecidable. □

The proof only holds when H must determine the status of every program and every input. It *is* possible to prove that a specific program with a specific input halts. For a sufficiently limited language, it is possible to solve the Halting Problem. For example, every finite program in a language without recursion or iteration must halt.

The theorem and proof can be extended to most observable properties of programs. For example, within the same structure one can prove that it is undecidable whether a program prints output or reaches a specific line in execution. Note that it is critical to the proof that $Q(P,x)$ does not actually run P; instead, it must decide what behavior P *would* exhibit, were it to be run, presumably by examining the source code of P. See http://www.cgl.uwaterloo.ca/ csk/halt/ for a nice explanation of the Halting Problem using the C programming language. Two straightforward ways to prove that a property is undecidable are:

- Show that the Halting Problem reduces to the property. That is, if you can solve static checking of the property, then you can solve the Halting Problem, therefore the property is at least as hard as the Halting Problem and is undecidable.
- Rewrite a Halting Problem proof, substituting the property for halting. Note that this is not the same as reducing the property to the Halting Problem.

Preview from Notesale.co.uk
Page 5 of 33

2.3 Significance of Decidability

Language designers are faced with a dilemma because, like halting, most properties of a program in a sufficiently powerful language are undecidable. One choice is to abandon checking certain properties until run time. We call run-time checking **dynamic checking**. The enables a language to easily express many kinds of programs, however, it means that the programmer has little assurance that the program is actually correct. Only extensive testing will find errors in such programs, and extensive testing is expensive, time consuming, and impractical for programs with large branch factors. Python and Scheme are two languages that defer almost all checking until run time. It is easy to write programs in these languages and hard to find errors in them.

Another choice is to restrict the expressive power of the language somewhat, so that more properties can be checked before a program is run. This is called **static checking** (a.k.a. **compile-time checking**). This makes it harder to construct programs, however it enables much stronger guarantees than testing can provide– the checker can *prove* that a program does not have certain kinds of errors, without ever running it. C++ and ML are languages that provide significant static checking facilities. A language like Java is somewhere in between. It performs some checks statically, but in order to make the language more flexible the designers made many other checks dynamic. The one of the most common check is the null-pointer dereference, which many programmers will recognize as their most common error as well. Java programmers may be pleasantly surprised to discover that there are many languages (unfortunately none of the popular ones mentioned in this paragraph) in which most null-pointer checks can be made statically, and therefore appear very infrequently as run-time errors.

A third choice is to restrict the expressive power of the language so severely that certain kinds of errors are simply not expressible. This is useful for metalanguages, like type systems, and for embedded languages, like

x^* = zero or more instances of x

x^+ = one or more instances of x

An example of these patterns for expressing a simple programming language literal expression domain (e.g., a subset of Scheme's literals):

$$
\begin{array}{rcl}
\textit{boolean} & ::= & \texttt{\#t} \mid \texttt{\#f} \\
\textit{digit} & ::= & \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \\
\textit{integer} & ::= & [\ \texttt{+} \mid \texttt{-}\ ]\ \textit{digit}^+ \\
\textit{rational} & ::= & \textit{integer}\ \texttt{/}\ \textit{digit}^+ \\
\textit{decimal} & ::= & [\ \texttt{+} \mid \texttt{-}\ ]\ \textit{digit}^*\ \texttt{.}\ \textit{digit}^+ \\
\textit{real} & ::= & \textit{integer} \mid \textit{rational} \mid \textit{decimal}
\end{array}
$$

BNF can be applied at both the **character level** (e.g., to describe a lexer/tokenizer) and the **token level** (e.g., to describe a parser). The preceding example operates on individual characters within a string and is useful to a tokenizer. An example of a subset of Scheme's expression domain represented in BNF at the token level:

$$
\begin{array}{rcl}
\textit{variable} & ::= & \textit{id} \\
\textit{let} & ::= & (\ \texttt{let}\ (\ [\ \textit{id}\ \textit{exp}\ ]^*\ )\ \textit{exp}\ ) \\
\textit{lambda} & ::= & (\ \texttt{lambda}\ (\ \textit{id}^*\ )\ \textit{exp}\ ) \\
\textit{exp} & ::= & \textit{variable} \mid \textit{let} \mid \textit{lambda}
\end{array}
\tag{3}
$$

Preview from Notesale.co.uk
Page 10 of 33

2.2 Syntactic Sugar

Some expressions make a language's syntax more convenient and compact without actually adding expressivity: they make the language sweeter to use. We say that an expression is **syntactic sugar** and adds no expressive power if it can be reduced to another expression with only local changes. That is, without rewriting the entire body of the expression or making changes to other parts of the program. Such expressions are also referred to as being **macro-expressive**. They can naturally be implemented entirely within the parser or as **macros** in languages with reasonable macro systems.

For example, in Java any `FOR` statement, which has the form:

`for (` *init* `;` *test* `;` *incr* `)` *body*

can be rewritten as a WHILE statement of the form:

init `; while (` *test* `) {` *body* *incr* `; }`

`FOR` therefore does not add expressivity to the language and is syntactic sugar. In Scheme, `LET` adds no power over `LAMBDA`, and `LET*` adds no power over `LET`. Java exceptions are an example of an expressive form that cannot be eliminated without completely rewriting programs in the language.

We can express the Java `FOR` loop's **reduction** in more formal notation as:

$$
\begin{array}{c}
\texttt{for}\ (\ \textit{exp}_{init}\ \texttt{;}\ \textit{exp}_{test}\ \texttt{;}\ \textit{exp}_{incr}\ )\ \textit{exp}_{body} \\
\Rightarrow \\
\textit{exp}_{init}\ \texttt{;}\ \texttt{while}\ (\ \textit{exp}_{test}\ )\ \{\ \textit{exp}_{body}\ \ \textit{exp}_{incr}\ \texttt{;}\ \}
\end{array}
\tag{4}
$$

Here the pattern on the left may be reduced to the simpler form on the left during evaluation. This is an example of a general mechanism for ascribing formal semantics to syntax that is described further in the following section.

eral, both a and b are reductions. Furthermore, there may be multiple conditions in a, notationally separated by spaces, that must all be true for the reduction(s) in b to be applied. These rules are useful for making progress when no other rule directly applies. For example, to evaluate the mathematical expression $1+(7+2)$, we must first resolve $(7+2)$. The rule for making progress on addition with nested subexpression on the right of the plus sign is:

$$\frac{exp_2 \Rightarrow num_2}{exp_1 \texttt{ + } exp_2 \Rightarrow exp_1 \texttt{ + } num_2} \tag{10}$$

which reads, "if expression #2 can be reduced to some number (by some other rules), then the entire sum can be reduced to the sum of expression #1 and that number." We of course need some way of reducing expressions on the *left* of the plus sign as well:

$$\frac{exp_1 \Rightarrow num_1}{exp_1 \texttt{ + } exp_2 \Rightarrow num_1 \texttt{ + } exp_2} \tag{11}$$

Applying combinations of these rules allows us to simplify arbitrarily nested additions to simple number additions.

There are two major interpreter designs: substitution and evaluation interpreters. Substitution interpreters transform expressions to expressions and terminate when the final expression is a literal. Their value domain is their literal domain. Evaluation/environment interpreters are implemented with an `EVAL` procedure that reduces expressions directly to values. Such a procedure recursively invokes itself, but program execution involves a single top-level `EVAL` call. These two models of implementation correspond to two styles of semantics:

1. **Small-step** operational semantics rules reduce expressions to **expressions** (like a substitution interpreter)
2. **Big-step** operational semantics rules reduce expressions to **values** (like an `EVAL` interpreter)

The following two subsections demonstrate how semantics are assigned in each. We use the context of a simple language that contains only single-argument procedure definition, variable, conditional, application, and booleans. Let these have the semantics of the equivalent forms in the Scheme language:

$$\begin{aligned} exp \quad ::= \quad & \texttt{(}\ \lambda\ \texttt{(}\ id\ \texttt{)}\ exp\ \texttt{)} \mid \\ & id \mid \\ & \texttt{(}\ if\ exp\ exp\ exp\ \texttt{)} \mid \\ & \texttt{(}\ exp\ exp\ \texttt{)} \mid \\ & \texttt{true} \mid \texttt{false} \end{aligned} \tag{12}$$

6.2 Small-step Example

6.2.1 Rules

Small-step operational semantics rules reduce expressions to expressions. Although the choice of implementation is not constrained by the style of semantics (much), small step maps most directly to a substitution-based interpreter (e.g., [Kri07, 15]). Under small-step semantics the value domain is merely the terminal subset of the expression domain, plus procedure values. That is, literals are values. It is useful to us later to define a subdomains in this definition:

$$\begin{aligned} ok \quad ::= \quad & \texttt{true} \mid \texttt{(}\ \lambda\ \texttt{(}\ id\ \texttt{)}\ exp\ \texttt{)} \\ val \quad ::= \quad & \texttt{false} \mid ok \end{aligned} \tag{13}$$

Those expressions require no progress rules because they are values. Variable expressions require no progress rules because variables are always substituted away by applications. Only conditional and application need be defined. Conditionals naturally have two obvious rules, that I name E-IfOk and E-IfFalse (the "E" stands for "Evaluate"):

Preview from Notesale.co.uk
Page 12 of 33

7.1 Syntax

The λ calculus is a language with surprisingly few primitives in the expression domain[10]:

$$\begin{array}{rcl} var & ::= & id \\ abs & ::= & \lambda\ id\ .\ exp \\ app & ::= & exp\ \ exp \\ exp & ::= & var \mid abs \mid app \mid (\ exp\ ) \end{array}$$

The last expression on the right simply states that parentheses may be used for grouping.

The language contains single value type, the single-argument procedure, in the value domain. In set notation this is:

$$val = proc = var \times exp$$

and in BNF:

$$val \quad ::= \quad \lambda\ id\ .\ exp$$

The abbreviated names used here and in the following discussions are mnemonics for: 'id' = 'identifier', 'abs' = 'abstraction' (since λ creates a procedure, which is an abstraction of computation), 'app' = 'procedure application', 'exp' = 'expression', 'proc' = 'procedure', and 'val' = 'value'.

Preview from Notesale.co.uk
Page 17 of 33

7.2 Semantics

The formal semantics are simply those of substitution [Pie02, 72]:

App-Part 1: (reduce the procedure expression towards a value)

$$\frac{exp_p \Rightarrow exp'_p}{exp_p exp_a \Rightarrow exp'_p exp_a} \tag{28}$$

App-Part 2: (reduce the actual parameter towards a value)

$$\frac{exp_a \Rightarrow exp'_a}{exp_p exp_a \Rightarrow exp_p exp'_a} \tag{29}$$

App-Abs: (apply a procedure to a value)

$$\lambda\ id\ .\ exp_{body}\ \ val\ \Rightarrow\ [id \rightarrowtail val]\, exp_{body} \tag{30}$$

The App-Abs rule relies on the same syntax for the *val* value and *abs* expression, which is fine in λ calculus because we're using pure textural substitution. In the context of a true value domain that is distinct from the expression domain, we could express it as an *abs* evaluation rule for reducing a procedure expression to a procedure value and and an application rule written something like:

$$\frac{val_p = \lambda\ id\ .\ exp_{body}}{val_p\ val_a \Rightarrow\ [id \rightarrowtail val_a]\, exp_{body}} \tag{31}$$

7.3 Examples

For the sake of giving simple examples, momentarily expand λ calculus with the usual infix arithmetic operations and integers. Consider the evaluation of the following expression:

$$\lambda\, x\, .\, (x+3)\ 7 \tag{32}$$

[10] This is specifically a definition of the *untyped* λ-calculus.

There are other algorithms that address some of these problems, like the **stop-and-copy** collector algorithm, which compacts all marked memory by moving objects around during collection, and **generational** collector algorithms that predict the lifetime of objects to minimize the size of the traced heap.

One interesting side note is **conservative collection**, which allows gc to operate on an implementation that does not provide run-time types. The way that it works is to treat *every* four bytes of allocated memory as if it were a legal pointer. Assuming block sizes are stashed right in front of the first byte of the block and that we have an allocated list, we can quickly reject many false pointers and for the rest can trace them through the system with mark-sweep. This is even more conservative than regular mark-sweep because sometimes four bytes will happen to match a valid block address when they are not in fact a pointer. Recall that mark-sweep could not determine whether a block will actually be used in the future, only whether it is referenced. So, the conservative collector is more conservative than regular mark-sweep, but neither is perfect. One popular implementation is the Boehm-Demers-Weiser collector http://www.hpl.hp.com/personal/Hans_Boehm/gc/, which is used to add garbage collection to C/C++.

Preview from Notesale.co.uk
Page 31 of 33