Modular Abstract Interpreters

Robby Findler

5 May 1995

Abstract

High-level programming languages allow more elegant and comprehensible expression of programs, but compilers for high-level languages can require sophisticated optimization to generate machine code that runs quickly. Optimizations are usually expressed in terms of some statically discoverable information, and abstract interpretation provides a good framework for discovering that information. Abstract interpretation is a framework for relating alternative semantics for a programming language to each other. However, abstract interpretation is difficult because one has to prove the correctness and termination of the abstract interpreters and each program analysis problem needs a different abstract interpreter.

I used a monadic programming style to define a framework for modular abstract interpreters in SML extended with extensible datatypes. Monadic programming is a style of programming that allows easy expression of modularity. The modularity simplifies the construction of the interpreters which reduces possibility of errors and makes it easy to add whatever analyses you want to your system.

I built the framework for the modular interpreters described in this thesis and implemented constant propagation in that framework.

Contents

1 Program Analysis

Program analysis is the problem of discovering invariants about the behavior of programs. An *invariant* of a program a property that holds for the program, regardless of the program's inputs. For example, in a certain loop, an invariant might be that the variable x has an integer value that is always one greater than the variable y.

Program analysis is particularly useful in the context of high-level programming languages. High-level programming languages make programming easier and therefore less error-prone, but can incur a performance penalty. They also allow more portability – because they have a large "semantic distance" from machine code. *Semantic distance* is an informal measure of how the semantics for two languages differ. The *semantics* of a programming language defines the "meaning" of programs. Often, they are given as rules about how the language's evaluation and any static checking the languages offers occur. For example, Scheme and SML have a relatively small semantic distance, especially compared to the distance between SML and assembly language. The large semantic distance between high-level programming languages and machine languages is what makes it hard to translate high-level programs into machine code and motivates program analysis.

Program analysis is also useful for understanding programs. Analysis can tell you things about your program that might not otherwise be obvious or easy to check. For example a type-checker can tell you what the type of any expression in your program is, assuming that the expression is type-correct. Program analysis can also help to find errors in programs. Type-checkers will reject programs that are not type-correct.

Program analysis is not easy to do. Many interesting invariants are undecidable, *e.g.*, the invariant cannot be computed because it depends on the input or it relies on knowing if an arbitrary piece of code terminates. The undecidability of the problem means that we can only hope for approximate solutions. Approximate solutions introduce another complication, namely, the notion of correctness of a solution is much more difficult to understand and analyze.

One of the classical problems in program analysis is the rule of signs problem. The *rule of signs* problem is where you try to find the sign of each number that is used in a program. Using this analysis you will approximate all numbers in the code with their signs, either $+$, $-$ or 0. You can begin this analysis by approximating all constants by their signs. Then, when you add two positive numbers, you know that the result must be positive. But, what happens when you add a negative and a positive number? Because you have deleted the magnitude information, you do not know if the result of that addition is going to be negative or positive. Different analysis solve this problem in different ways.

There are several good approaches to program analysis. Dataflow analysis[Hec77] is a form of program analysis that can be easily computed and gives lots of interesting information to an optimizer. Dataflow analysis computes the live range of each "variable." In dataflow analysis terminology, a *variable* is a value that the program computes. Each variable, in normal terminology, corresponds to one location in memory and may be overwritten many times. In dataflow analysis each time the location in memory is written over, the old variable is killed and a new one is created. Dataflow analysis computes a def-use chain for each variable. A *def-use chain* is a chain that links each use of a variable to its creation, or definition. In dataflow terminology a *definition* of a variable is when the variable is created.

For example, in the language C a variable may be defined at the top of a function. That definition does not correspond with what the dataflow analysis definition of a variable and that C variable may in fact correspond to several different dataflow variables, `i.e.` in this C program:

```
void main () {
 int x;
 x = 1;
 x = x + 1;
 return x;
}
```

there is one C definition of a variable, but there are two dataflow variables. When x is written over with $x + 1$, a new dataflow variable is created and the first dataflow variable is said to be *killed*.

A def-use chain is useful for many optimizations. For example if you know that a variable is being defined inside a loop and the variables that are part of its definition are defined outside of the loop, you can hoist the definition of the variable outside the loop. Dataflow analysis can also be used to eliminate dead code. If you know that a variable's def-use chain has only the definition and no uses, than you do not need to compute the variable.

Type-inference[Mil78] is another example of program analysis. A type-inference algorithm determines what the type for each variable (in the usually sense of a variable) in a program, and rejects programs whose types do not make sense. Type-inference also allows the programmer to omit definition of variables. He or she may just bind variables to expressions and the type-inference algorithm will determine the type and hence the storage requirements for that variable. For example, in SML, the expression

$$\mathbf{fn}\ x \Rightarrow x + 1$$

would be transformed so that all of the variables had types attached to them. This particular expression would become:

$$\mathbf{fn}\ x{:}int \Rightarrow x + 1$$

and the type of the whole expression would be $int \rightarrow int$.

A third example of program analysis is abstract interpretation[CC77]. Abstract interpretation is a framework for relating two different semantics of a programming language. Furthermore, the alternative semantics are written in such a way that one of them can be statically computed. Then using abstract interpretation you can formalize the connection between the two semantics.

2 Abstract Interpretation

For this thesis I chose to use abstract interpretation as a framework for program analysis. Abstract analysis subsumes many other analyses of programs. It is a very general framework that applies to a wide range of program analyses. The Cousots[CC77] popularized abstract interpretation and put it on a sound semantic footing. The description of abstract interpretation that I give here is a simplification from the general framework. See [CC77] for a more general treatment.

Abstract interpretation is a framework for relating alternative semantics for a programming language to the standard semantics for the programming language. First you start with the standard semantics, which is a mapping that accepts a program $p \in P$ from the set of all programs, and a value $d \in \mathcal{D}$ from the domain of values and returns a new value, $e \in \mathcal{D}$, and is written $[\![p]\!]d = e$. Generally $[\![\cdot]\!]$ will not be a computable function. We want to define an *abstract semantics* that is both computable and related in a useful way to the standard semantics.

We define an abstraction. An abstraction is a tuple of four values, $\langle \{\!| \cdot |\!\}, \alpha, \gamma, \hat{D} \rangle$, where $\hat{D} \subseteq \mathcal{P}(\mathcal{D})$, $\{\!| \cdot |\!\}$ is a mapping for all programs p, $\{\!|p|\!\} : \hat{D} \to \hat{D}$, $\alpha : \mathcal{P}(\mathcal{D}) \to \hat{D}$, is called the *abstraction function* and $\gamma : \hat{D} \to \mathcal{P}(\mathcal{D})$ is called the *concretization function* satisfying these properties:

1. $\{\!|p|\!\}$ is monotonic and computable for all $p \in P$,
2. $\hat{D}$ is a complete lattice and has $\top$ and $\bot$, and infimum and supremum of $\hat{D}$ under the lattice ordering.
3. the range of $\{\!|p|\!\}$ is a sub-lattice of $\hat{D}$ and has no infinite incresing chains,
4. $\alpha(\gamma(\hat{d})) \subseteq \hat{d}$, for all $\hat{d} \in \hat{D}$,
5. $\gamma(\alpha(d)) \supseteq d$, for all $d \in \mathcal{P}(\mathcal{D})$, and
6. α and γ are increasing and computable.

Define the *collecting semantics* $(\!|p|\!) : \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$ for all $p \in P$ to be:

$$(\!|p|\!)D = \{[\![p]\!](d) | d \in D\}$$

Theorem 1 *if $\hat{d}$ is a fixed point of $\{\!|p|\!\}$ then $(\!|p|\!)(\gamma(\hat{d})) \subseteq \gamma(\{\!|p|\!\}(\hat{d}))$ for all $p \in P$.*

This theorem means that the abstract semantics is conservative, which is to say when you concretize a fixed point of the abstract semantics and then apply the normal semantics to all of the members in that set, the concretization of what the abstract semantics found either the same set of answers or possibly more answers, but never fewer. So, in some sense, you have covered all of the bases when you abstractly interpret something, you have computed a conservative approximation to the correct answer.

Assuming the proof that statement, we can now find an upper bound, D, on the set of states that are reachable from any given input, d, and program p given by:

$$D = \gamma(\text{lfp}(\{\!|p|\!\}))(d)$$

where lfp is a function that finds the least fixed point of a function on some input.

From property (1) we know that any one application of $\{\!|p|\!\}$ is computable, and from property (3) we know that all chains in the range of $\{\!|p|\!\}$ are finite so we can compute the least fixed point by repeated applications of $\{\!|p|\!\}$ to the input d. Also, by theorem (1) we know that $[\![p]\!](d) \in D$. Depending on how good you abstract interpretation is you may have that $D = \mathcal{D}$, which is to say that you have gained no information, but hopefully you will have found some interesting information.

For example if we chose this simple language with lambda abstractions and numbers:

E :: $L \mid N$.

L :: $\lambda x.E \mid E\ E \mid x$.

x :: `variable names`.

N :: $E + E \mid E - E \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid \ldots$.

and our standard semantics were this:

$$\begin{aligned}
[\![(\lambda x.e_1)e_2]\!]\ \rho\ i &= [\![e_1]\!]\ ([x \hookrightarrow ([\![e_2]\!]\ \rho\ i)]\rho)\ i \\
[\![\lambda x.e]\!]\ \rho\ i &= [\![e]\!]\ ([x \hookrightarrow i]\rho)\ i \\
[\![x]\!]\ \rho\ i &= \rho(x) \\
[\![e_1 + e_2]\!]\ \rho\ i &= [\![e_1]\!]\ \rho\ i + [\![e_2]\!]\ \rho\ i \\
[\![e_1 - e_2]\!]\ \rho\ i &= [\![e_1]\!]\ \rho\ i - [\![e_2]\!]\ \rho\ i \\
[\![n]\!]\ \rho\ i &= n
\end{aligned}$$

Note that there are several expression that can be constructed by the grammar that do not appear in the definition of the semantics. These expression are considered illegal and evaluation of them is nonsensical, and not defined. Also, note that ρ wasn't in the general theory of abstract interpretation. ρ here represents an environment which can have bindings added to, for example if you were to bind x to v in ρ it would be written: $[x \hookrightarrow v]\rho$, and to lookup x in and environment ρ is written $\rho(x)$. The fact that ρ is missing from the general theory is unimportant because if you just pass the environment with no bindings in it to initialize these standard semantics you will have the same thing as the standard semantics in the general theory. The same idea holds for the abstract semantics defined later in this section. I also have not specified eager or lazy evaluation of function arguments in these semantics. Conceivably the environment could compute the value when the environments bind variables or when they look up variables.

Going back to the example of the rule of signs if you wanted to compute the rule of signs invariant, you would first define the lattice of values, $\hat{D}$, with $\perp$ as the lowest element in the lattice, $\top$ as the highest element in the lattice and $+$, $-$ and 0 as elements that are unrelated to each other, but all lower than $\top$ and higher than $\perp$, as in figure (1), where if x is below y in the picture means that $x < y$.

γ and α are defined:

$$\alpha(x) = \begin{cases} 0 & \text{if } x = 0 \\ + & \text{if } x > 0 \\ - & \text{if } x < 0 \end{cases} \qquad \gamma(x) = \begin{cases} \mathbb{Z} & \text{if } x = \top \\ 0 & \text{if } x = 0 \\ \mathbb{P} & \text{if } x = + \\ -\mathbb{P} & \text{if } x = - \\ \emptyset & \text{if } x = \perp \end{cases}$$

Define ϱ to be an environment where all variables are bound to $\top$. Then your abstract semantics is defined:

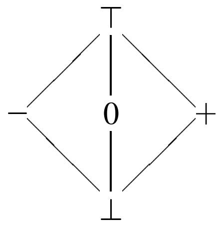

Figure 1: Rule of Signs Lattice

$$
\begin{array}{rcl}
\{\!|\lambda x.e|\!\}\,\rho\; i & = & \{\!|e|\!\}\,\varrho\; i \\
\{\!|(\lambda x.e_1)e_2|\!\}\,\rho\; i & = & \{\!|e_2|\!\}\,\varrho\; i \\
\{\!|x|\!\}\,\rho\; i & = & \rho(x) \\
\{\!|e_1 + e_2|\!\}\,\rho\; i & = & \{\!|e_1|\!\}\,\rho\; i + \{\!|e_2|\!\}\,\rho\; i \\
\{\!|e_1 - e_2|\!\}\,\rho\; i & = & \{\!|e_1|\!\}\,\rho\; i - \{\!|e_2|\!\}\,\rho\; i \\
\{\!|n|\!\}\,\rho\; i & = & \begin{cases} - & \text{if } n < 0 \\ 0 & \text{if } n = 0 \\ + & \text{if } n > 0 \end{cases}
\end{array}
$$

where $+$ and $-$ are defined on abstract values exactly as you would expect, *e.g.* when you add or subtract a negative and a positive number you get $\top$, but when you add two positive numbers you get a positive number, when you add two negative numbers you get a negative number, and so on.

Note that this abstraction is very conservative. Whenever a function application happens, all information is lost because of the use of ϱ. The abstraction will terminate, but will only tell you information about lambda-less terms.

Abstract interpretation bas been successfully used for alias analysis[Deu92], strictness analysis[BHA86], binding time analysis[BHA86, JGB^{+}90, Con90] and many other program analysis problems.

3 Modular Interpreters

People have been writing interpreters for a long time. They are easy to write when compared to compilers, and because of the relative ease of implementation they are used to rapidly prototype languages. They allow one to experiment with new language ideas quickly, and they are portable.

For example, for the language defined in section (2) supported lambda abstractions and the natural numbers:

$$E \;::\; L \mid N.$$

$$L \;::\; \lambda x.E \mid E\; E \mid x.$$

x :: `variable names.`

N :: $E + E \mid E - E \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid \ldots.$

Using, SML a call-by-name interpreter could be written:

```
datatype term =
    Apply of term * term | Lambda of string * term | Var of string
  | Num of int | Sum of term * term | Difference of term * term

exception Error of string

fun eval env (Apply (e,f)) = let
        val e = eval env e
    in
        case e of
          Lambda (s,b) => eval (bind env (s,f) b)
        | _ => raise Error "illegal application"
    end
  | eval env (Lambda (s,b)) = Lambda (s,b)
  | eval env (Var s) = lookup env s
    handle Lookup => raise Error "unbound variable"
  | eval env (Num n) = (Num n)
  | eval env (Sum (i,j)) = let
        val i = eval env i
        val j = eval env j
    in
        case (i,j) of
          (Num n,Num m) => Num (n+m)
        | _ => raise Error "non-number added"
    end
  | eval env (Difference (i,j)) = let
        val i = eval env i
        val j = eval env j
    in
        case (i,j) of
          (Num n,Num m) => Num (n-m)
        | _ => raise Error "non-number subtracted"
    end

val eval = eval empty
```

Modular interpreters make experimentation with different features of a programming language and how they interact easy to accomplish. It also allows the programmer to concentrate on each interpreter separately hopefully leading to fewer mistakes in coding.

For example if you were to modularize the above interpreter, you might separate the function into two parts:

```
datatype term1 = Apply of term * term | Lambda of string * term
               | Var of string

exception Error of string

fun eval1 env (Apply (e,f)) = let
        val e = eval1 env e
    in
        case e of
          Lambda (s,b) => eval1 (bind env (s,f) b)
        | _ => raise Error "illegal application"
    end
  | eval1 env (Lambda (s,b)) = Lambda (s,b)
  | eval1 env (Var s) = lookup env s
    handle Lookup => raise Error "unbound variable"

val eval1 = eval1 empty

datatype term2 =  Num of int | Sum of term * term
               | Difference of term * term

exception Error of string

fun eval2 env (Num n) = (Num n)
  | eval2 env (Sum (i,j)) = let
        val i = eval2 env i
        val j = eval2 env j
    in
        case (i,j) of
          (Num n,Num m) => Num (n+m)
        | _ => raise Error "non-number added"
    end
  | eval2 env (Difference (i,j)) = let
        val i = eval2 env i
        val j = eval2 env j
    in
        case (i,j) of
          (Num n,Num m) => Num (n-m)
        | _ => raise Error "non-number subtracted"
    end

val eval2 = eval2 empty
```

Then you would combine the two parts into one evaluator. Defining how to combine two modules of an interpreter into an interpreter is what makes this problem hard. Deciding on an abstraction boundary between modules of a language is a difficult task. To separate out different features of a programming language into orthogonal modules can sometimes be impossible and finding the right compromise between generality and easy composition of the different features can be difficult.

Liang Sheng, Mark Jones and Paul Hudak[LHJ95] devised an approach to modular interpreters in Gofer[Jon94]. They used Gofer's user-defined overloading and type-classes and a monadic programming style to define each module of the interpreter. Steele[GLS94] used Haskell[Deu92] and monads to define his modular interpreters, relying heavily on the laziness of Haskell to tie his modules together.

4 SML

SML[MTH90] is a programming language that supports static polymorphic type-checking with datatypes, pattern matching and an expressive module system. The major points of interest for this thesis are the datatypes and the module system.

4.1 Datatypes

Datatypes provide a way for users of SML to define their own types. A datatype consists of a number of *constructors*, representing a disjoint union, each of which may or may not carry with it a value of a (possibly recursive) type. Datatypes can also be polymorphic, which is to say that constructors can carry values whose types are not specified at the definition of the datatype. Constructors can then be pattern matched against input to functions. For example, a datatype that represented colors could be written:

datatype *color* = *Red* | *Blue* | *Green*

Declaring the color datatype in SML leads to the introduction of three constructors to the environment, namely *Red*, *Blue* and *Green*. Each of these constructors is a value of type *color* and maybe pattern matched against in a function declaration. Function declarations in SML are usually written as a case-wise analysis of the input type. So to write a function that mapped from the color datatype, you could write:

fun *favorite Red* = *false*
  | *favorite Blue* = *true*
  | *favorite Green* = *false*

The function *favorite* has the type $color \rightarrow bool$ and it says that my favorite color is blue. Colors might also have an intensity along with them, and the following definition encapsulates the intensities of the colors.

datatype *color* = *Red* **of** *int*
                | *Blue* **of** *int*
                | *Green* **of** *int*

Now the my favorite function is written:

$$\begin{array}{l}\textbf{fun}\ \mathit{favorite}\ (\mathit{Red}\ \_) = \mathit{false}\\ \quad |\ \mathit{favorite}\ (\mathit{Blue}\ n) = n > 30\ \textbf{andalso}\ n < 50\\ \quad |\ \mathit{favorite}\ (\mathit{Green}\ \_) = \mathit{false}\end{array}$$

This definition means that my favorite color blue has intensity between 30 and 50.* Datatypes are also recursive. A tree datatype and a function that flattened a tree in an inorder traversal could be written like this: (where "@" is a concatenation operator on lists and "[" and "]" build list objects)

$$\begin{array}{l}\textbf{datatype}\ \text{'}a\ \mathit{tree} = \mathit{Leaf}\ \textbf{of}\ \text{'}a\\ \qquad\qquad\qquad\quad |\ \mathit{Node}\ \textbf{of}\ \text{'}a * \text{'}a\ \mathit{tree} * \text{'}a\ \mathit{tree}\\ \textbf{fun}\ \mathit{flatten}\ (\mathit{Leaf}\ \mathit{value}) = [\mathit{value}]\\ \quad |\ \mathit{flatten}\ (\mathit{Node}\ (\mathit{value},\mathit{left},\mathit{right})) = (\mathit{flatten}\ \mathit{left})@[\mathit{value}]@(\mathit{flatten}\ \mathit{right})\end{array}$$

As you can see the *flatten* function has two cases. The first case is when the tree consists of only a leaf node, and the inorder traversal of the nodes is just the singleton list containing the value at that leaf, *value*. When the tree has a node, the list containing the inorder traversal of the nodes has the nodes in the left tree, concatenated onto the singleton list containing this node, concatenated onto the nodes in the right subtree.

4.2 Module System

SML's module system consists of structures, signatures and functors. A *structure* is an environment of values, types and other structures. A *signature* is a specification of a structure. For example a signature that specified a structure that handled addition could be written like this:

$$\begin{array}{l}\textbf{signature}\ \mathit{ADDITION}{=}\textbf{sig}\\ \quad \textbf{type}\ \mathit{num}\\ \quad \textbf{val}\ \mathit{add} : \mathit{num} * \mathit{num} \rightarrow \mathit{num}\\ \quad \textbf{val}\ \mathit{zero} : \mathit{num}\\ \quad \textbf{val}\ \mathit{one} : \mathit{num}\\ \quad \textbf{val}\ \mathit{zerop} : \mathit{num} \rightarrow \mathit{bool}\\ \textbf{end}\end{array}$$

This signature specifies an environment that includes a type of natural numbers and an addition function on them. *num* is the type of natural numbers in the *ADDITION* signature, *add* is a function that takes two natural numbers, adds them together and returns the result. *zero* and *one* are just the numbers 0 and 1, represented as *num*'s, and *zerop* is a function that tests to see if a number is zero. Note that "*" is the product type and "→" is the function type in SML.

Signatures can also be refinements of other signatures. Intuitively speaking, a signature is a *refinement* of a another signature if all of the values and types in the first signature are also in the second signature.

One way to define refinements of signatures is to use the **include** command. For example the *MULTIPLICATION* signature is a refinement of the *ADDITION* signature. Any structure that matched the *MULTIPLICATION* signature would also match the *ADDITION* signature.

*I made up these numbers. They probably do not correspond to reality at all.

```
signature MULTIPLICATION=sig
  include ADDITION
  val mult : num * num -> num
end
```

You can build multiple structures that match the same signature, each with different properties. For example a trivial structure matching the *ADDITION* signature would be:

```
structure Int:ADDITION=struct
  type num=int

  fun add (a,b)=a+b

  fun minus1 0=0
    | minus1 n=n-1

  fun zerop 0=true
    | zerop n=false

  val zero=0
  val one=1
end
```

Here the type of numbers is just the integers. Addition of integers is SML's normal addition. Similarly *minus1*, *zerop*, *zero* and *one* are defined in terms of the existing integers in SML.

A more complicated structure that supports a very rudimentary bignums is:[†]

```
structure BigUnary:ADDITION=struct
  datatype digit = One
  type num = digit list

  fun add (l,r) = l@r

  fun minus1 []=[]
    | minus1 (One::rest)=rest

  fun zerop [] = true
    | zerop _ = false

  val zero=[]
  val one=[One]
end
```

In *BigUnary* numbers are represented in unary, as a list of digits. To add two numbers, you just concatenate their digit lists. Two subtract one from a number, you just pluck a digit off of the front of the number. *zerop* checks to see if the digit list is the empty list. *zero* and *one* are the empty list and the singleton digit list respectively.

A slightly less rudimentary bignum package would be:

[†]Calling this representation a BigNum package is a little misleading. In SML, integers are 31 bit unsigned quantities. In a reasonable implementation of SML, each element in a list takes two words, so the number $2^{30} - 1$ takes approximately 2^{31} words or 2^{11} megabytes of memory. This is unreasonable even if you assume that SML has nothing else loaded and no garbage collector, *i.e.* it SML will allocate until it runs out of memory.

```
structure BigBinary:ADDITION=struct
  datatype binary_digit = One | Zero
  type num = binary_digit list

  fun add (Zero::rest,Zero::rest)=Zero::(add (rest,rest))
    | add (One::rest,Zero::rest)=One::(add (rest,rest))
    | add (One::rest,One::rest)=add (Zero::(add (rest,rest)),[One])
    | add ([],n)=n
    | add (n,[])=n

  fun trim_trailing l=
      let fun helper (Zero::rest)=helper rest
            | helper (n as One::rest)=n
      in helper (rev l) end
  fun minus1 []=[]
    | minus1 (One::rest)=Zero::rest
    | minus1 (Zero::rest)=trim_trailing (One::(minus1 rest))

  fun zerop [] = true
    | zerop _ = false

  val zero=[]
  val one=[One]
end
```

In *Big_Binary*, numbers are represented in binary. A digit is either a one or a zero and a number is a list of digits. The most significant digits are at the beginning of the list and the least significant are at the end. There are now five cases for addition and addition is defined recursively. This structure demonstrates an interesting fact about structures. Notice that the structure has a function, *trim_trailing* which trims zeros off of the front of the list, *i.e.* it would transform the number "000110" to the number "110." This function is not visible outside the declaration of the structure, because it is not mentioned in the signature and therefore is not allowed to escape to users of the structure. All three of these structures present a uniform interface to natural numbers.

A *functor* is a structure that is parameterized on another structure. For example this functor builds a *MULTIPLICATION* structure from an *ADDITION* structure.

```
functor Add_Mult(structure A:ADDITION):MULTIPLICATION=
struct
  open A
  fun mult (a,b)=if (zerop a)
                   then zero
                   else add (b,mult (minus1 a,b))
end
```

The *Add_Mult* structure accepts an *Add_Mult* structure and adds a multiplication function. The multiplication is defined very naively, but works with all three different *ADDITION* structures. The **open** statement opens a structure, combining the environments in the addition structure and the environment in the resultant multiplication structure.

5 Modular Abstract Interpreters

I used an extended version of SML, one with extensible datatypes. An extensible datatype is a datatype for which you can declare new constructors at any time in your program. I used Sheng's[LHJ95] basic framework to define my modular interpreters, but in a different context. Each module of the abstract interpreter must match a *MODULE* signature and then can be easily combined with other *MODULE*'s to form larger interpreters.

5.1 Individual Modules

Monads are key in the definition of the interpreters. A monad has three important components, the type of monads, *unitM* and *bindM*. Their use and definition have been well documented in [Mog91a, Mog91b, LHJ95, Wad92]. Here is the signature for a monad:

```
signature BIND_MONAD=sig
  type 'a M
  val unitM : 'a -> 'a M
  val bindM : ('a M * ('a -> 'b M)) -> 'b M
  val showM : 'a M -> 'a
end
```

The *MODULE* signature is designed to be easy to write structures for and at the same time provide enough information to allow easy arbitrary combination of different *MODULES*.

```
signature MODULE=sig
  structure M:BIND_MONAD

  exception Error of string

  extensible datatype term
  extensible datatype anorm
  extensible datatype value

  type state = (Id.id,value) Poly_Env.env

  exception Step_Unknown (* this exception should be raised when this
                            step function is asked to step on part of
                            the language it is not responsible for. *)
  val step : (anorm * state -> (state * Id.id) M.M) ->
             anorm * state -> (state * Id.id) M.M

  exception Eq_Unknown (* this exception should be raised when this ==
                          function is asked to check equality on part
                          of the language it is not responsible
                          for. *)
  val == : (value * value -> bool) -> value * value -> bool

  val parse : (term,char list,'a) Parser.P -> (term,char list,'a) Parser.P
```

```
    val Anorm_Unknown (* this exception should be raised when this anorm
                        function is asked to anormalize part of the
                        language it is not responsible for. *)
    val anorm : ((string,Id.id) Poly_Env.env * term * Id.id → anorm → anorm) →
                (string,Id.id) Poly_Env.env * term * Id.id → anorm → anorm

    exception Unparse_Unknown
    val unparse : (anorm → string) → anorm → string
  end
```

The general style of the type of each of the functions in this signature is $(\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$ and the idea is that each function accepts as its first argument the whole function and then evaluates the part of the domain for which it is responsible, calling its first argument to recurse. Each function also has an exception that it should raise when it is asked to evaluate part of the domain for which it is not responsible.

At the heart of each module of the modular abstract interpreter are two functions, *step* and ==. *step* takes a step in the evaluation of a language, and == compares two values for equality. *step*'s first argument is a function to call when it recurses and then it accepts a *state* and a *term* and returns a new *state* and an identifier. A *state* is a finite map that takes program points to their values. The *step* function should look at the term that it receives, evaluate it and bind it's program point to the result of the evaluation, returning the new state and the program point that was just bound.

parse and *anorm* are helper functions for the evaluator. *parse* parses text into abstract syntax and *anorm* A-Normalizes[FSDF93] the abstract syntax tree. A-Normalization is a process which inserts a name for every program point in the program and renames all of the variables so that no two names are the same. Having a name for every program points makes many analyses much more useful.

parse is written using Chris Okasaki's[‡] combinator parsing library. The structure of Okasaki's parsers is already analogous to a function, and so the type of each *MODULE*'s parser is a function that accepts a parser for the entire language and then returns a parser that will parse the section of language that this *MODULE* represents.

When an abstract syntax tree has been anormalized it looks like a large let statement, where each different value that would be computed by the evaluation of the abstract syntax tree is bound to a variable, and values are the only things that can be bound to variables. *anorm* is defined with a specific incantation in mind. Ignoring its first argument, *anorm*, takes an environment mapping from the input text's variable names to the generated variable name, a term to be translated and an identifier to bind the term, and returns a function. That function takes an value of type *anorm* and puts it in a hole that it made while anormalizing the given term. So, for example, if this function were anormalizing a term that was just the number 1, it would take the environment, take the term, and an identifier, and return a function that would take an anormalized term and would return an anormalized term where the identifier is bound to 1 in the input term.

5.2 Pairing Modules

The *Pair* functor combines two modules into a single module.

[‡] `cokasaki+@cs.cmu.edu`

```
functor Pair (structure E:MODULE
              structure F:MODULE
              sharing E.M=F.M
              and type E.term=F.term
              and type E.anorm=F.anorm
              and type E.value=F.value):MODULE=
struct
  exception Step_Unknown
  exception Eq_Unknown
  exception Anorm_Unknown
  exception Unparse_Unknown
  exception Error of string
  type state=E.state
  type term=E.term
  type anorm=E.anorm
  type value=E.value
  structure M=E.M
  val step = fn step => fn e =>
    ((E.step step e) handle E.Step_Unknown => (F.step step e))
      handle E.Error s => raise Error s
           | F.Error s => raise Error s
           | F.Step_Unknown => raise Step_Unknown
  val == = fn == => fn p =>
    ((E.== == p) handle E.Eq_Unknown => (F.== == p))
      handle E.Error s => raise Error s
           | F.Error s => raise Error s
           | F.Eq_Unknown => raise Eq_Unknown
  val anorm = fn anorm => fn s => fn a =>
    ((E.anorm anorm s a) handle E.Anorm_Unknown => (F.anorm anorm s a))
      handle E.Error s => raise Error s
           | F.Error s => raise Error s
           | F.Anorm_Unknown => raise Anorm_Unknown

  val unparse = fn unparse => fn a =>
    ((E.unparse unparse a) handle E.Unparse_Unknown => (F.unparse unparse a))
      handle E.Error s => raise Error s
           | F.Error s => raise Error s
           | F.Unparse_Unknown => raise Unparse_Unknown
  open Parser
  infix 1 ||
  fun parse a = E.parse a || F.parse a
```

Each of the component pairing functions do exactly the same thing. First they check module *E* and if *E* does not handle the input, they check module *F*, each being careful to handle *E*'s and *F*'s exceptions and raise the new exceptions. Combination of two parsers uses the function || which means "or."

5.3 Tying Off Modules

When you have paired together all of your modules you will want to transform the resultant module into something useful. A possibility for that is this *LANGUAGE* signature.

```
signature LANGUAGE=sig
  type anorm
  type value
  type state = (Id.id,value) Poly_Env.env
  exception Error of string
  val print : anorm → unit
  val parse : string → anorm
  val eval : anorm → state ∗ value
end
```

The *LANGUAGE* signature provides a reasonable final user-level interface to the language that you have built. It includes a *parse* function that accepts program text and returns an anormalized representation of the source's abstract syntax tree, an *eval* function that evaluates the anormalized abstract syntax and it returns a final value and a state which is a mapping from program points to the values that were taken on in the evaluation, and finally a print function for displaying the anormalized input. This *Tie_Off* functor maps from structures matching the *MODULE* signature into ones matching *LANGUAGE*.

```
functor Tie_Off (structure E:MODULE
                 val default:E.value option):LANGUAGE=
struct
  type anorm=E.anorm
  type value=E.value
  type state=E.state
  type term=E.term
  exception Error=E.Error
  fun helper (e,e') = E.== helper (e,e')
  infix 3 ==
  fun s == s' = Poly_Env.== helper (s,s')
  fun small_steps (AB.Id id,s) = E.M.unitM (s,id)
    | small_steps (t,s) = E.step small_steps (t,s)
  fun big_step (t,s) =
    E.M.bindM (small_steps (t,s),
               fn (s',id) ⇒ if s == s' then E.M.unitM (s',id)
                                        else big_step (t,s'))
  fun eval t = let
    val (s,id) = E.M.showM (big_step (t,Poly_Env.empty (Id.cmp,default)))
  in
    (s,Poly_Env.lookup s id)
  end
```

```
    val pref = ref Parser.fail : (term,char,term Stream.T) Parser.P ref
    val p = E.parse (Parser.$(fn () => !pref))
    val _ = (pref := p)

    fun parse s = let
      val term=Stream.head ((Parser.parses p)
                           (Position.markstream (Input.readstring s)))
      val id=Id.genid "final" (* id is the innermost binding and the
                                 result of the whole computation. *)
      fun cmp (a,b)=if a=b then LibBase.Equal else
        if String.<(a,b) then LibBase.Less else LibBase.Greater
    in
      anorm (Poly_Env.empty (cmp,NONE),term,id) (AB.Id id)
    end

    fun unparse (AB.Id x) = (UPI.indent ())^(Id.makestring x)
      | unparse a = E.unparse (fn x => "\n"^(UPI.indent ())^(unparse x)) a

    val print = print o (fn x=>x^"\n") o unparse

    fun anorm a = E.anorm anorm a

  end
```

When I defined each of the components of a *MODULE*, I specified functions of the form $(\alpha \to \beta) \to \alpha \to \beta$ and to tie those off, each of the functions in the *Tie_Off* functor has something like this at it's heart:

```
  fun f a = E.f f a
```

The tied off function passes itself into the function in the module. The only recursion in the definition of each of the tied off functions is only happening here in the *Tie_Off* functor.

The input the functor is *E* is the module to be tied off, and a default value for bindings in the state. If it is *NONE* there will be no default values, and if it is *SOME x* then it will be *x*.

First I define *helper* which the equality function on all values in the structure *E*. Then I use the *Poly_Env* equality function to define an equality function on states, ==. *Poly_Env*'s equality function accepts an equality function on values that have the same type as the range of the enviornment and returns a function that compares enviornments for equality.

Next I define the *small_steps* function. *small_steps* ties off the *step* function from *E*. *big_step* uses the *small_steps* function and iterates application of the *small_steps* function, testing to see if the state is the same before and after the application of *small_steps*. Finally *eval* uses *big_step* to find the final result and the final state.

pref, *p*, and *parser* are all used to define the parser. *pref* is a ref cell to the final parser, which gets filled in with a dummy default value: the parser that fails on any input. Then using the "$" function provided with the parser, which will apply unit to the function that it is given at the "right" time, we can tie the knot for the parser.

The unparser also handles the identifier case, and provides some support for controlling indentation in the *AB* structure.

5.4 Arithmetic Module

```
signature NUMBERS_MODULE=
sig
  include MODULE
  constructor term = Int of int
                   | Add of state * state
                   | Multiply of state * state

  constructor value = Int of int
  constructor anorm = LetAdd of {id:Id.id,i:Id.id,j:Id.id,a:anorm}
                    | LetMultiply of {id:Id.id,i:Id.id,j:Id.id,a:anorm}
                    | LetInt of {id:Id.id,i:int,a:anorm}
end
```

The *NUMBERS_MODULE* signature is a refinement of the *MODULE* signature. It adds new abstract syntax terms, new anormalized terms and a value for handling numbers. Notice in the new constructors *LetAdd* and *LetMultiply* for the type *anorm* that there is a value of type *id* which marks this program point, a pair of identifiers which represent two values for *i* and *j* and *a* which is the body in which the sum or product is bound.

The *Numbers* functor implements these operations using SML's built in integers. The *Poly_Env* structure is the environment structure that is used throughout the code.

```
functor Numbers(structure M:BIND_MONAD
                type anorm
                type value
                type term):NUMBERS_MODULE=
struct
  structure P=Poly_Env
  type term = term
  type value = value
  type state = (Id.id,value) P.env
  type anorm = anorm
  structure M = M

  constructor term = Int of int
                   | Add of state * state
                   | Multiply of state * state

  constructor value = Int of int
  constructor anorm = LetAdd of {id:Id.id,i:Id.id,j:Id.id,a:anorm}
                    | LetMultiply of {id:Id.id,i:Id.id,j:Id.id,a:anorm}
                    | LetInt of {id:Id.id,i:int,a:anorm}
```

The header of the functor just defines the types and adds the constructors to the correct types as specified by the *NUMBERS_MODULE* signature.

```
val step = fn step ⇒ fn
    (LetInt {id,i,a},s) ⇒ step (a,P.bind ((id,Int i),s))
  | (LetAdd {id,i,j,a},s) ⇒
      (case (P.lookup s i,P.lookup s j) of
          (Int x,Int y) ⇒ step (a,P.bind ((id,Int (x+y)),s))
        | _ ⇒ raise Error "Non-number added")
  | (LetMultiply {id,i,j,a},s) ⇒
      (case (P.lookup s i,P.lookup s j) of
          (Int x,Int y) ⇒ step (a,P.bind ((id,Int (x*y)),s))
        | _ ⇒ raise Error "Non-number multiplied")
  | _ ⇒ raise Step_Unknown

  val == = fn == ⇒ let infix == in
    fn (Int x,Int y) ⇒ x=y
     | (Int _,_) ⇒ false
     | (_,Int _) ⇒ false
     | _ ⇒ raise Eq_Unknown
  end
```

The *step* function uses the monadic programming style. In each case it calls itself recursively on *a* after binding *id* to an appropriate value, either the result of the addition, multiplication or just the integer value. If *step* receives a value that it is not defined for, it raises the *Step_Unknown* exception to signal that another module should handle the evalutaion.

The == function compares two integers for equality by using the built-in SML integer equality function and if it is called with an integer and something that is not an integer it returns false. If it receives input that does not contain any integers, it raises the *Eq_Unknown* exception to signal that another module should be responsible for checking equality.

```
fun anorm_ anorm (e,Int i,id) a = LetInt {id=id,i=i,a=a}
    | anorm_ anorm (e,Add (i,j),id) a = let
        val id_left=Id.genid "add_left"
        val id_right=Id.genid "add_right"
      in
        anorm (e,i,id_left)
       (anorm (e,j,id_right)
       (LetAdd {id=id,i=id_left,j=id_right,a=a}))
       end
    | anorm_ anorm (e,Multiply (i,j),id) a = let
          val id_left=Id.genid "add_left"
          val id_right=Id.genid "add_right"
        in
          anorm (e,i,id_left)
        (anorm (e,j,id_right)
        (LetMultiply {id=id,i=id_left,j=id_right,a=a}))
        end
    | anorm_ _ _ _ = raise Anorm_Unknown

  val anorm = anorm_
```

The function *anorm_* defines the anormalization of the aritmetic module. To anormalize an integer, you just translate to the *LetInt* case of the type *anorm*. To anormalize addition and multiplication, the function must first anormalize the arguments and then bind those identifiers with either the *Let_Add* or the *Let_Multiply* constructors. Since identifiers can only be bound to values, all of the computation must have been split up and named in the recursion.

```
val unparse = fn unparse ⇒
    fn LetMultiply {id,i,j,a} ⇒
       "letmultiply "^(Id.makestring id)^" = "^(Id.makestring i)^" * "^
       (Id.makestring j)^(unparse a)
     | LetAdd {id,i,j,a} ⇒
       "letadd "^(Id.makestring id)^" = "^(Id.makestring i)^" + "^
       (Id.makestring j)^(unparse a)
     | LetInt {id,i,a} ⇒
       "letint "^(Id.makestring id)^" = "^(makestring i)^(unparse a)
     | _ ⇒ raise Unparse_Unknown

  open Parser Parsing_Utils
  infix  2 -- ##
  infixr 3 &&
  infix  2 wth suchthat return guard
  infixr 1 ||

  fun ok_char x=x <= #"9" andalso x >= #"0"
  fun maybe f (NONE,x)=x
    | maybe f (SOME x,y)=f(x,y)

  fun parse p = let
    fun repeat2 p = p && repeat1 p wth op::
    val p_w=second whitespace p
  in
      opt (literal #"~") && repeat1 (any suchthat ok_char)
      wth Int o sint_to_int o (maybe op::)
   || middle (literal #"(") (second (literal #"+") (repeat2 p_w)) (literal #")")
      wth (fn x::xs ⇒ fold Add xs x | _ ⇒ raise Error "parse add")
   || middle (literal #"(") (second (literal #"*") (repeat2 p_w)) (literal #")")
      wth (fn x::xs ⇒ fold Multiply xs x | _ ⇒ raise Error "parse multiply")
  end
end
```

To print out the anormalized form of an expression, the *unparse* function first prints out an appropriate string designating the case and then the identifier followed by the sub expression of the term. The parser recognises a simple scheme-like syntax.

5.5 Lambda Module

The lambda module adds in the ability to define and use lambda abstractions.

```
signature LAMBDA_MODULE=
sig
  include MODULE
  type var=string
  eqtype tag
  constructor value = Lambda of tag * var * anorm
  constructor term = TermLambda of var * term
                   | App of term * term
                   | LambdaVar of var
  constructor anorm = LetLambda of {id:Id.id,v:var,body:anorm,a:anorm}
                    | LetApp of {id=Id.id,f:Id.id,arg:Id.id,a:anorm}
                    | LetLambdaVar of {id=Id.id,v=var,a=anorm}
end
```

The *LAMBDA_MODULE* signature adds one new value and three new terms into the language for dealing with lambda abstractions. Lambda values are tagged to be able to compare them for equality. The tagging scheme is not specified by the signature.

```
functor Lambda_Module(structure M:ENV_MONAD
                      type term type anorm type value
                      sharing type M.Env.range=anorm
                          and type M.Env.domain=string):LAMBDA_MODULE=
struct
  type term=term
  type anorm=anorm
  type value=value
  type state = (Id.id,value) Poly_Env.env
  exception Step_Unknown
  exception Eq_Unknown
  exception Anorm_Unknown
  exception Unparse_Unknown
  exception Error of string
  type var=string
  structure M=M
  structure P=Poly_Env
  type tag = Id.id (* the tags are the positions in the text where the
                      lambdas occur *)
  constructor value = Lambda of tag * var * anorm
  constructor term = TermLambda of var * term
                   | App of term * term
                   | LambdaVar of var
  constructor anorm = LetLambda of {id:Id.id,v:var,body:anorm,a:anorm}
                    | LetApp of {id=Id.id,f:Id.id,arg:Id.id,a:anorm}
                    | LetLambdaVar of {id=Id.id,v=var,a=anorm}
```

The top of the functor defines the types, exceptions and constructors required for the lambda module. Note that the tags are represented by identifiers and will be the source positions of the lambda definitions. Using these tags for equality, two lambdas will be equal if and only if they are from the same place in the source. Also, a structure that matches *ENV_MONAD* is required by the lambda module, not just a regular *BIND_MONAD*.

```
signature ENV_MONAD=
sig
  include BIND_MONAD

  structure M:BIND_MONAD
  structure Env:MONOMORPHIC_ENVIORNMENT

  val rdEnv : Env.env M
  val inEnv : Env.env → 'a M → 'a M
  val clEnv : Env.range → Env.env M

  datatype 'a env_M = E of Env.env → 'a M.M
  sharing type M=env_M
end
```

The signature *ENV_MONAD* provides three important operations in addition to the usual monadic operations[§]. The *Env* structure inside the *ENV_MONAD* is a monomorphic enviornment that provides the background for the enviornment monad. *rdEnv* provides a way to get the current enviornment, *inEnv* provides a way to evaluate something in an enviornment and *clEnv* clears an enviornment, replacing the everything in the range with its first argument.

```
val step = fn step ⇒
    fn (LetLambdaVar {id,v,a},s) ⇒
        (M.bindM (M.rdEnv, fn env ⇒
                  step (a,P.bind ((id,M.Env.lookup env v),s)))
         handle M.Env.Lookup ⇒ raise Error ("unbound var: "^v))
     | (LetLambda {id,v,body,a},s) ⇒
        step (a,P.bind ((id,Lambda (id,v,body)),s))
     | (LetApp {id,f,arg,a},s) ⇒
        (case (P.lookup s f) of
            Lambda (_,v,body) ⇒ M.bindM
              (M.rdEnv, fn env ⇒ M.inEnv
               (M.Env.bind env (v,P.lookup s arg))
               (M.bindM (step (body,s),fn (s,res_id) ⇒
                         step (a,P.bind ((id,P.lookup s res_id),s)))))
          | _ ⇒ raise Error "application of non-function"
              handle P.Lookup ⇒ (print "handled P.Lookup in letapp\n";
                                 raise P.Lookup))
     | _ ⇒ raise Step_Unknown
```

[§]the idea for this monad comes from [LHJ95]

```
val == = fn == => let infix ==
in
  fn (Lambda (id,_,_),Lambda (id',_,_)) => id=id'
   | (Lambda _,_) => false
   | (_,Lambda _) => false
   | _ => raise Eq_Unknown
end
```

The *step* function handles binding of variables bound by lambda abstractions by using a special monad. The monad keeps track of which variables are bound to which values during the evaluation of a term. The evaluate a lambda variable, you just look up the variable in the enviornment and bind *id* to that value. To evaluate a lambda, you just bind the lambda to *id* and continue. To evaluate an application, you first check to see if the value in the function position is a lambda. If so, you bind the argument variable to the argument and evaluate the lambda body, binding the result to *id* and continue on evaluating *a*. If the value in the function position is not a lambda, then raise an error exception. If any other term comes into *step* then *step* raises the *Step_Unknown* exception.

The == function checks to see if two lambdas have the same tag. If they do, the lambdas are equal and if they do not, the two lambdas are not equal. == also checks if something other than a lambda is compared to a lmabda, than the two values are not equal.

```
fun anorm_ anorm (e,TermLambda (v,t),id) a = let
        val body_id=Id.genid "lambda body"
      in
        LetLambda {id=id,v=v,body=anorm (e,t,body_id) (AB.Id body_id),a=a}
      end
   | anorm_ anorm (e,LambdaVar v,id) a = LetLambdaVar {id=id,v=v,a=a}
   | anorm_ anorm (e,App (t,t'),id) a = let
        val f_id=Id.genid "fun half of app"
        val arg_id=Id.genid "arg half of app"
      in
        anorm (e,t,f_id)
        (anorm (e,t',arg_id)
         (LetApp {id=id,f=f_id,arg=arg_id,a=a}))
      end
   | anorm_ _ _ _ = raise Anorm_Unknown

val anorm = anorm_
```

```
val unparse = fn unparse ⇒
  fn LetLambdaVar {id,v,a} ⇒
     "letlambdavar "^(Id.makestring id)^" = "^v^(unparse a)
   | LetLambda {id,v,body,a} ⇒ let
       val _ =UPI.go_out()
       val fore="letlambda "^(Id.makestring id)^" = lambda "^v^"."
       val body=unparse body
       val _=UPI.go_in ()
       val aft=unparse a
     in
       fore^body^aft
     end
   | LetApp {id,f,arg,a} ⇒
     "letapp "^(Id.makestring id)^" = "^(Id.makestring f)^" "^
     (Id.makestring arg)^(unparse a)
   | _ ⇒ raise Unparse_Unknown

open Parser Parsing_Utils
  infixr 3 &&
  infix  2 -- ## wth suchthat return guard
  infixr 1 ||

  fun parse p = let
    val p_w=second whitespace p
    val l_id = literal #"l" && repeat1 (any suchthat is_alphabetic)
               wth (implode o op::)
  in
      (middle (Parser.string (explode "(lambda"))
              (middle whitespace l_id whitespace)
              (literal #".")) &&
      (first p_w (literal #")")) wth TermLambda
    || l_id wth LambdaVar
    || middle (literal #"(") (p && (second whitespace p)) (literal #")") wth App
  end
end
```

The anormalization and parsing of the lambdas is very much like the anormalization and parsing in the numbers module.

5.6 Constant Propagation

Constant propagation is an analysis which looks for constants additions and multiplications and tries to generate enough information to propagate them. It is guaranteed to terminate and will try to provide information about variables that are bound to constants or sums and products of constants. The structure for this particular constant propagator is a direct translation of Christopher Colby's[Col] constant propagator.

```
signature CONSTPROP_MODULE=sig
  include MODULE
```

```
  constructor value = Top | Bottom
end
```

The *CONSTPROP_MODULE* adds two new values, *Top* and *Bottom*. *Top* represents an unknown value. If a variables takes on more than one possible value during the evaluation of a program, that variables will be bound to top in the state that the constant propogator returns. *Bottom* represents variables that are not reached in the execution of a program. It is the default value that all variables are bound to when execution begins.

```
functor ConstProp(structure M:BIND_MONAD
                  structure N:NUMBERS_MODULE
                  type anorm type term type value):CONSTPROP_MODULE=
struct
  exception Step_Unknown
  exception Eq_Unknown
  exception Anorm_Unknown
  exception Unparse_Unknown

  type term=term
  type anorm=anorm
  type value=value

  exception Error of string
  type state = (Id.id,value) Poly_Env.env
  structure M=M

  exception Top
  exception Bottom

  val == = fn _ ⇒ fn (Top,Top)         ⇒ true
                  | (_,Top)            ⇒ false
                  | (Top,_)            ⇒ false
                  | (Bottom,Bottom)    ⇒ true
                  | (Bottom,_)         ⇒ false
                  | (_,Bottom)         ⇒ false
                  | _                  ⇒ raise Eq_Unknown

  fun step _ _ = raise Step_Unknown
  fun anorm _ _ _ = raise Anorm_Unknown
  fun unparse _ _ = raise Unparse_Unknown
  fun parse _ = Parser.fail

end
```

Bottom and *Top* do not appear in the source language of the program, so *anorm*, *parse*, and *unparse* do not do anything. Since there are no new constructors for the type *anorm* introduced, *step* does not need to do anything either. The only function here is ==, which just says that *Top* is equal to *Top* and *Bottom* is equal to *Bottom*.

```
functor ConstProp_Numbers (structure M:BIND_MONAD
                           structure CP:CONSTPROP_MODULE
                           structure N:NUMBERS_MODULE):NUMBERS_MODULE=
struct
  structure P=Poly_Env
  open N

  val step = fn step => fn
     (LetInt {id,i,a},s) => step (a,P.bind ((id,Int i),s))
   | (LetAdd {id,i,j,a},s) =>
       (case (P.lookup s i,P.lookup s j) of
            (Int x,Int y) => step (a,P.bind ((id,Int (x+y)),s))
          | (CP.Top,Int _) => step (a,P.bind ((id,CP.Top),s))
          | (Int _,CP.Top) => step (a,P.bind ((id,CP.Top),s))
          | (CP.Top,CP.Top) => step (a,P.bind ((id,CP.Top),s))
          | (CP.Bottom,_) => step (a,P.bind ((id,CP.Bottom),s))
          | (_,CP.Bottom) => step (a,P.bind ((id,CP.Bottom),s))
          | _ => raise Error "Non-number added")
   | (LetMultiply {id,i,j,a},s) =>
       (case (P.lookup s i,P.lookup s j) of
            (Int x,Int y) => step (a,P.bind ((id,Int (x*y)),s))
          | (CP.Top,Int _) => step (a,P.bind ((id,CP.Top),s))
          | (Int _,CP.Top) => step (a,P.bind ((id,CP.Top),s))
          | (CP.Top,CP.Top) => step (a,P.bind ((id,CP.Top),s))
          | (CP.Bottom,_) => step (a,P.bind ((id,CP.Bottom),s))
          | (_,CP.Bottom) => step (a,P.bind ((id,CP.Bottom),s))
          | _ => raise Error "Non-number multiplied")
   | _ => raise Step_Unknown

end
```

The *ConstProp_Numbers* functor accepts a structure, *N*, which matches the *NUMBERS_MODULE* signature. The functor then uses *N*'s to provide the parsing and anormalization functionality, and the only function left to define is a new step that computes the invariant, in this case which constants can be propagated.

To propagate constants for the numbers module of a language, we just do the addition or multiplication normally if we are adding or multiplying integers. If we try to add *Top* and an integer, we know that the first number is undetermined, so the sum must also be undetermined. If we add *Bottom* to another integer, we know that we can never reach *Bottom* and hence can never reach the sum.

```
functor ConstProp_Lambda(structure M:ENV_MONAD
                         structure CP:CONSTPROP_MODULE
                         structure L:LAMBDA_MODULE
                         type t = (Id.id,L.value) Poly_Env.env
                           and type L.state = t
                           and type L.tag = Id.id
                           and type M.Env.domain=L.var
                              and M=L.M):LAMBDA_MODULE=
struct
  structure P=Poly_Env
  structure M'=M
  open L
  structure M=M'
  val all_top = P.empty (Id.cmp,SOME CP.Top)
  val step = fn step =>
    fn (LetLambda {id,v,body,a},s) =>
         step (a,Poly_Env.bind ((id,Lambda (id,v,body)),s))
     | (LetLambdaVar {id,v,a},s) =>
         (M.bindM (M.rdEnv, fn env =>
                   step (a,P.bind ((id,M.Env.lookup env v),s)))
          handle M.Env.Lookup => raise Error ("unbound var: "^v))
     | (LetApp {id,f,arg,a},s) =>
         ((case (P.lookup s f) of
             Lambda (_,v,body) => M.bindM
               (M.clEnv CP.Top, fn env => M.inEnv env
                (M.bindM (step (body,all_top),
                         fn (s',res_id) => step (a,s))))
           | CP.Top => step (a,P.bind ((id,CP.Top),s))
           | _ => raise Error "application of non-function")
           handle P.Lookup => (print "handled P.Lookup in letapp\n";
                               raise P.Lookup))
     | _ => raise Step_Unknown
end
```

To propagate constants through lambda expressions we have to be careful to avoid non-termination. When evaluating an application, on both entry to and exit from a lambda expression, I replace all bindings to the environment with *Top*, including the argument to the lambda expression. And, *Top* applied to anything is *Top*. This scheme avoids non-termination and gives us information about constants that does not span function calls.

6 Extensible Datatypes

6.1 Why extensible datatypes?

In an unextended SML it is difficult to modular interpreters in an elegant way. As a first attempt one might try a module like this:

signature *MODULE*=**sig**
  type *term*
  type *value*
  val *eval* : *term* $\rightarrow$ *value*
end

However, you cannot tie these together in an easy way. When you try to plug them together you find that you need the finished interpreter before you start to tie off the recursion. Steele[GLS94] does something similar to this, except that he uses Gofer, a lazy language and is able to express this easily because the language is lazy.

A more complicated attempt that also does not work:

signature *MODULE*=**sig**
  type *term*
  type *value*
  val *eval* : (*term* $*$ *value* *option* *CML.chan*) *CML.chan*
end

The idea here is that you can use CML[Rep91] channels and instantiate all of the eval threads at once, and then at some later time spawn a process to watch each of them. You would then have an evaluator that relied heavily on concurrency. This doesn't work because of the interactions between weak polymorphism and the SML module system.

6.2 Example, Concrete Syntax

Currently in SML if you define a datatype, you must specify all of its constructors at once, *e.g.*

datatype *units* = *Inches* | *Feet* | *Yards*

fun *to_inches* *n* *Inches* = *n*
  | *to_inches* *n* *Feet* = *n*$*$12
  | *to_inches* *n* *Yards* = *to_inches* (*n*$*$3) *Feet*

With the addition of extensible datatypes, you can add additional constructors to datatypes anytime that you want, so for example we could add the constructor *miles* to the datatype *units* like this:

constructor *units* = *Miles*
fun *to_inches*' *n* *Miles* = *to_inches* (*n*$*$5280) *Feet*
  | *to_inches*' *n* *u* = *to_inches* *n* *u*

The function *to_inches'* now handles all of the previously defined units as well as *Miles*.

6.3 Semantics

I wrote out the static and dynamic semantics for a non-polymorphic fragment of SML extended with extensible datatypes.

I let E range over expressions, τ over types, Δ over environments mapping from identifiers to their types, Γ over environments mapping from identifiers that define type to the types they define and x over variables in the language. $\Box$ is used to represent a unique identifier, and Ξ the set from which the $\Box$'s are drawn. Each time a new $\Box$ is introduced through an introduction of a datatype, each $\Box$ is different than the others that preceded it. Rules that begin with typed- are part of the static semantics, and rules that begin with ev- are part of the dynamic semantics.

$$\frac{\Gamma + \{x \mapsto \Box\}, \Delta \vdash E{:}\tau \qquad \tau \neq \Box}{\Gamma, \Delta \vdash \texttt{let extensible datatype } x \texttt{ in } E \texttt{ end}{:}\tau}\text{typed-defty-dt}$$

The typed-defty-dt rule introduces extensible datatypes. Each extensible datatype is marked with a unique identifier. No two extensible datatypes are the same, even if they have the same constructors.

$$\frac{\Gamma \vdash E \hookrightarrow V}{\Gamma \vdash \texttt{let extensible datatype } x \texttt{ in } E \hookrightarrow V}\text{ev-defty}$$

During the evaluation of extensible datatypes, I just ignore the datatype definition and evaluate the body of the let where the datatype is bound.

$$\frac{\Gamma(C) = \langle \tau_d, \tau_c \rangle \qquad \Gamma, \Delta \vdash E{:}\tau_d}{\Gamma, \Delta \vdash C \;\; E{:}\tau_c}\text{typed-construct}$$

Constructors are represented as a pair in the type environment, where the first projection is the type of values that the constructor carries, and the second projection is the datatype that the constructors are a part of. If an expression types to τ_d then the application of the constructor to that expression has type τ_d.

$$\frac{\Gamma \vdash E \hookrightarrow V}{\Gamma \vdash C \;\; E \hookrightarrow C \;\; V}\text{ev-construct}$$

To evaluate the application of a constructor to an expression, I first evaluate the expression to a value and then I wrap the value with the constructor.

$$\texttt{constructor } x \texttt{ = } C \texttt{ of } \tau.$$

To add a constructor to the datatype x, use the `constructor` statement. This adds a constructor C carrying the type τ to the datatype x.

$$\frac{\begin{array}{c}\Gamma(x) = \tau_c \qquad \tau_c \in \Xi \\ \tau \neq \langle \tau_d, \tau_c \rangle \qquad \Gamma + \{C \mapsto \langle \tau_d, \tau_c \rangle\}, \Delta \vdash E{:}\tau\end{array}}{\Gamma, \Delta \vdash \texttt{let constructor } x \texttt{ = } C \texttt{ of } \tau_d \texttt{ in } E \texttt{ end}{:}\tau}\text{typed-dt-extend}$$

To type-check an extension of a datatype, first I must make sure that x is a datatype. Remember that Ξ is the set of all $\Box$'s. I must also check to make sure that τ_d is not a constructor, but is a type. Then I bind C to the pair $\langle \tau_d, \tau_c \rangle$ in Γ and continue type-checking the body.

$$\frac{\Gamma \vdash E \hookrightarrow V}{\Gamma \vdash \texttt{let constructor } x \texttt{ = } C \texttt{ of } \tau \texttt{ in } E \hookrightarrow V}\text{ev-dt-extend}$$

To evaluate an extension of a datatype, just ignore the extension. Any application of this constructor to a value in the body of E is legal if the expression typechecked, so I can just throw away the definition of the constructor.

$$\texttt{destruct } E \texttt{ of } C\ x \texttt{ => } E \mid x \texttt{ => } E.$$

To destruct a value from a constructor, use the `destruct` statement. `destruct` will check to see if the expression E is constructed with the constructor C. If so, it will bind x to the value inside the constructor and take the first branch. If E was not created with the constructor C then the second branch will be taken, with x bound to the expression E.

$$\frac{\begin{array}{cc}\Gamma(C) = \langle \tau_d, \tau_c \rangle & \Gamma, \Delta \vdash E{:}\tau_c \\ \Gamma, \Delta + \{x_1 \mapsto \tau_d\} \vdash E_1{:}\tau & \Gamma, \Delta + \{x_2 \mapsto \tau_c\} \vdash E_2{:}\tau\end{array}}{\Gamma, \Delta \vdash \texttt{ destruct } E \texttt{ of } C\ x_1 \texttt{ => } E_1 \mid x_2 \texttt{ => } E_2{:}\tau}\text{typed-destruct}$$

To type-check the `destruct` statement, first need to make sure that C is a constructor. Then I need to make sure that E is a member of the type that C constructors values for, *i.e.* τ_c. In the first case, x_1 should have the type of the values that C carries and in the second case x_2 will have the same type as E. Both cases of the destruct statement must type to the same type, τ and the entire expression also types to τ.

$$\frac{\Gamma \vdash E \hookrightarrow C\ \ V' \qquad \Gamma + \{x_1 \mapsto V'\} \vdash E_1 \hookrightarrow V}{\Gamma \vdash \texttt{destruct } E \texttt{ of } C\ x_1 \texttt{ => } E_1 \mid x_2 \texttt{ => } E_2 \hookrightarrow V}\text{ev-destruct-match}$$

In the case where E evaluates to an application of the constructor C to a value, bind x to the value and evaluate E_1.

$$\frac{\begin{array}{c}\Gamma \vdash E \hookrightarrow C'\ \ V' \qquad C' \neq C \\ \Gamma + \{x_2 \mapsto C'\ \ V'\} \vdash E_2 \hookrightarrow V\end{array}}{\Gamma \vdash \texttt{destruct } E \texttt{ of } C\ x_1 \texttt{ => } E_1 \mid x_2 \texttt{ => } E_2 \hookrightarrow V}\text{ev-destruct-mismatch}$$

If E evaluates to a constructor applied to a value that is different from the constructor mentioned in the `destruct` statement, bind x_2 to E and evaluate E_2

6.4 Elf Semantics

I implemented the above semantics in Elf[Pfe91]. Elf is a logical programming language that allows proofs and semantics to be expressed and checked. Elf uses a technique called higher order abstract syntax, which allows you to use elf variables to represent the object language variables, in this case the non-polymorphic mini-ML's variables.

Here is the definition of a datatype:

```
dt-inst : cn t.
```

Each different datatype has it's own name. Because of the use of higher order abstract syntax each name for the datatype is a unique identifier for the datatype. So, if two datatypes have the same constructors they will not be equal. Datatypes are defined with no constructors to start, and constructors are added later using the `constructor` judgement.

```
constructors : type.
constructor : names % the name of the constructor
   -> cn t % the type of the domain
   -> cn t % the datatype for which this is a constructor
   -> constructors.
```

A constructor is specified by its name, the type of the values that it carries and the extensible datatype that it maps into.

I also proved the type and value soundness for the language. The formal statement of value soundness is:

Theorem 2 *for any expression e, evaluation either doesn't terminate or yields a value.*

which means that the using the dynamic semantics that I wrote down, if any program written in the language terminates, it's final result is a lambda abstraction, an integer, a boolean or a character.

The statement of type soundness is:

Theorem 3 *for any expression e, if e is of type τ and e evaluates to v then v has type τ.*

and means that if the type system infers a type for some expression and the evaluator finds a value for the expression, the value will have the same type that the original value had. The proofs for both theorems are in sections (B.6) and (B.7).

7 Conclusion

I added extensible datatypes to SML. I worked out the semantics for a monomorphic fragment of the core of SML with extensible datatypes and proved type-soundness and value-soundness for that fragment. I worked out a framework for the definition of modular interpreters and then extended that to work for modular abstract interpreters. I then implemented the standard semantics for a language with lambdas, lets, and integer constants. I also implemented a constant propagation algorithm using the framework.

The addition of extensible datatypes to SML allows for a flexible and useful system of modular abstract interpreters to be developed. Each module of the interpreter is an SML structure which is easily combined with any other modules of the interpreter to build complicated abstract interpreters. The composability of these modules has been shown[LHJ95] to be very flexible and it's extension to abstract interpretation is natural and should be able to support abstract definition over the range of the expressions in any language.

While the definition of the abstract interpreter seems very natural from the definition of the standard interpreters, more work should be done to experimentally verify that different abstract interpretations are indeed expressible in this framework. There are two reasons why abstract interpretation is hard. One is because the interpreters must be proved to terminate and be correct, and the other is because each analysis requires a different abstract interpretation. Some work has been done to prove that if the modules are correct and terminate separately, they will terminate and be correct when they are combined. I believe that the way these modules have been designed the reusability of them will help with the problem of having to redesign a brand-new abstract interpreter from scratch each time, but only implementing more abstract interpreters in this framework can demonstrate that conclusively.

8 Acknowledgements

Thanks to Peter Lee for advising me this year. Also thanks to Chris Okasaki who answered myriads of little questions I had.

A Semantics

A.1 Grammar

This is a grammar representing the syntax of the language.

$E ::\ N \mid B \mid P \mid F \mid T \mid x \mid E\texttt{:}\tau \mid \texttt{let}\ I\ \texttt{in}\ E\ \texttt{end} \mid \texttt{(}E\texttt{)}.$

$x ::\ \texttt{variable names.}$

$C ::\ \texttt{constructor names.}$

$\tau ::\ \texttt{nat} \mid \texttt{bool} \mid \tau\ \texttt{*}\ \tau \mid \tau\ \texttt{->}\ \tau \mid x.$

$I ::\ x\ \texttt{=}\ E \mid x\texttt{:}\tau\ \texttt{=}\ E \mid \texttt{type}\ x\ \texttt{=}\ \tau$
$\quad \mid \texttt{extensible datatype}\ x \mid \texttt{constructor}\ x\ \texttt{=}\ C\ \texttt{of}\ \tau.$

$N ::\ \texttt{0} \mid \texttt{s}\ E \mid \texttt{case}\ E\ \texttt{of 0 =>}\ E \mid \texttt{s}\ x\ \texttt{=>}\ E.$

$B ::\ \texttt{true} \mid \texttt{false} \mid E\ \texttt{and}\ E \mid E\ \texttt{or}\ E \mid \texttt{not}\ E$
$\quad \mid \texttt{if}\ E\ \texttt{then}\ E\ \texttt{else}\ E.$

$P ::\ \texttt{(}E\texttt{,}\ E\texttt{)} \mid \texttt{fst}\ E \mid \texttt{snd}\ E.$

$F ::\ \texttt{lam}\ x\ \texttt{=>}\ E \mid \texttt{lam}\ x\texttt{:}\tau\ \texttt{=>}\ E \mid E\ E \mid \texttt{fix}\ x\ \texttt{=>}\ E.$

$T ::\ C\ E \mid \texttt{destruct}\ E\ \texttt{of}\ C\ x\ \texttt{=>}\ E \mid x\ \texttt{=>}\ E.$

A.2 Static Semantics

These are the static semantics for mini-ML with extensible datatypes. We let E range over expressions, τ over types, Δ over environments mapping from identifiers to their types, Γ over environments mapping from identifiers that define type to the types they define and x over variables in the language. $\Box$ is used to represent a unique identifier, and Ξ the set from which the $\Box$'s are drawn. Each time a new $\Box$ is introduced through an introduction of a datatype, each $\Box$ is different than the others that preceded it.

$$\frac{}{\Gamma,\Delta\vdash \texttt{z}{:}\texttt{nat}}\,\textsf{typed-z} \qquad \frac{\Gamma,\Delta\vdash E{:}\texttt{nat}}{\Gamma,\Delta\vdash \texttt{s}\ \ E{:}\texttt{nat}}\,\textsf{typed-s}$$

$$\frac{\Gamma,\Delta\vdash E_1{:}\texttt{nat} \qquad \Gamma,\Delta\vdash E_2{:}\tau \qquad \Gamma,\Delta+\{x\mapsto\texttt{nat}\}\vdash E_3{:}\tau}{\texttt{case}\ \ E_1\ \ \texttt{of}\ \ \texttt{0}\ \ \texttt{=>}\ \ E_2 \mid\ \texttt{s}\ \ x\ \ \texttt{=>}\ \ E_3{:}\tau}\,\textsf{typed-case}$$

$$\frac{}{\Gamma,\Delta\vdash \texttt{true}{:}\texttt{bool}}\,\textsf{typed-true} \qquad \frac{}{\Gamma,\Delta\vdash \texttt{false}{:}\texttt{bool}}\,\textsf{typed-false}$$

$$\frac{\Gamma,\Delta\vdash E_1{:}\texttt{bool} \qquad \Gamma,\Delta\vdash E_2{:}\texttt{bool}}{\Gamma,\Delta\vdash E_1\ \ \texttt{and}\ \ E_2{:}\texttt{bool}}\,\textsf{typed-and}$$

$$\frac{\Gamma,\Delta\vdash E_1{:}\texttt{bool} \qquad \Gamma,\Delta\vdash E_2{:}\texttt{bool}}{\Gamma,\Delta\vdash E_1\ \ \texttt{or}\ \ E_2{:}\texttt{bool}}\,\textsf{typed-or}$$

$$\frac{\Gamma,\Delta\vdash E{:}\texttt{bool}}{\Gamma,\Delta\vdash \texttt{not}\ \ E{:}\texttt{bool}}\,\textsf{typed-not}$$

$$\frac{\Gamma,\Delta\vdash E_t{:}\texttt{bool} \qquad \Gamma,\Delta\vdash E_1{:}\tau \qquad \Gamma,\Delta\vdash E_2{:}\tau}{\Gamma,\Delta\vdash \texttt{if}\ \ E_t\ \ \texttt{then}\ \ E_1\ \ \texttt{else}\ \ E_2{:}\tau}\,\textsf{typed-if}$$

$$\frac{\Gamma,\Delta\vdash E_1{:}\tau_1 \qquad \Gamma,\Delta\vdash E_2{:}\tau_2}{\Gamma,\Delta\vdash \texttt{(}E_1\texttt{,}E_2\texttt{)}{:}\tau_1\ \ \texttt{*}\ \ \tau_2}\,\textsf{typed-pair}$$

$$\frac{\Gamma,\Delta\vdash E{:}\tau_1\ \ \texttt{*}\ \ \tau_2}{\Gamma,\Delta\vdash \texttt{fst}\ \ E{:}\tau_1}\,\textsf{typed-fst} \qquad \frac{\Gamma,\Delta\vdash E{:}\tau_1\ \ \texttt{*}\ \ \tau_2}{\Gamma,\Delta\vdash \texttt{snd}\ \ E{:}\tau_2}\,\textsf{typed-snd}$$

$$\frac{\Gamma,\Delta\vdash x{:}\tau_d \qquad \Gamma,\Delta+\{x\mapsto\tau_d\}\vdash E{:}\tau_c}{\Gamma,\Delta\vdash \texttt{lam}\ \ x\ \ \texttt{=>}\ \ E{:}\tau_d\ \ \texttt{->}\ \ \tau_c}\,\textsf{typed-lam}$$

$$\frac{\Gamma,\Delta\vdash \texttt{lam}\ \ x\ \ \texttt{=>}\ \ E{:}\tau_d\ \ \texttt{->}\ \ \tau_c}{\Gamma,\Delta\vdash \texttt{lam}\ \ x\texttt{:}\tau_d\ \ \texttt{=>}\ \ E{:}\tau_d\ \ \texttt{->}\ \ \tau_c}\,\textsf{typed-clam}$$

$$\frac{\Gamma,\Delta\vdash E_1{:}\tau_d\ \ \texttt{->}\ \ \tau_c \qquad \Gamma,\Delta\vdash E_2{:}\tau_d}{\Gamma,\Delta\vdash E_1\ \ E_2{:}\tau_c}\,\textsf{typed-app}$$

$$\frac{\Gamma,\Delta\vdash \texttt{lam}\ \ x\ \ \texttt{=>}\ \ E{:}\tau\ \ \texttt{->}\ \ \tau}{\Gamma,\Delta\vdash \texttt{fix}\ \ x\ \ \texttt{=>}\ \ E{:}\tau}\,\textsf{typed-fix} \qquad \frac{\Gamma,\Delta\vdash \texttt{fix}\ \ x\ \ \texttt{=>}\ \ E{:}\tau}{\Gamma,\Delta\vdash \texttt{fix}\ \ x\texttt{:}\tau\ \ \texttt{=>}\ \ E{:}\tau}\,\textsf{typed-cfix}$$

$$\frac{E{:}\tau}{E\texttt{:}\tau{:}\tau}\,\textsf{typed-cexp}$$

$$\frac{\Gamma,\Delta \vdash E_1{:}\tau_d \qquad \Gamma,\Delta + \{x \mapsto \tau_d\} \vdash E_2{:}\tau_c}{\Gamma,\Delta \vdash \texttt{let}\ x\ \texttt{=}\ E_1\ \texttt{in}\ E_2\ \texttt{end}{:}\tau_c}\text{typed-let}$$

$$\frac{\Gamma,\Delta \vdash E_1{:}\tau_d \qquad \Gamma,\Delta + \{x \mapsto \tau_d\} \vdash E_2{:}\tau_c}{\Gamma,\Delta \vdash \texttt{let}\ x\texttt{:}\tau_d\ \texttt{=}\ E_1\ \texttt{in}\ E_2\ \texttt{end}{:}\tau_c}\text{typed-clet}$$

$$\frac{\Gamma + \{x \mapsto \tau'\},\Delta \vdash E{:}\tau}{\Gamma,\Delta \vdash \texttt{let type}\ x\ \texttt{=}\ \tau'\ \texttt{in}\ E\ \texttt{end}{:}\tau}\text{typed-defty-ty}$$

$$\frac{\Gamma + \{x \mapsto \Box\},\Delta \vdash E{:}\tau \qquad \tau \neq \Box}{\Gamma,\Delta \vdash \texttt{let extensible datatype}\ x\ \texttt{in}\ E\ \texttt{end}{:}\tau}\text{typed-defty-dt}$$

$$\frac{\Gamma(C) = \langle \tau_d, \tau_c \rangle \qquad \Gamma,\Delta \vdash E{:}\tau_d}{\Gamma,\Delta \vdash C\ \ E{:}\tau_c}\text{typed-construct}$$

$$\frac{\begin{array}{c}\Gamma(C) = \langle \tau_d, \tau_c \rangle \qquad \Gamma,\Delta \vdash E{:}\tau_c \\ \Gamma,\Delta + \{x_1 \mapsto \tau_d\} \vdash E_1{:}\tau \qquad \Gamma,\Delta + \{x_2 \mapsto \tau_c\} \vdash E_2{:}\tau\end{array}}{\Gamma,\Delta \vdash\ \texttt{destruct}\ E\ \texttt{of}\ C\ x_1\ \texttt{=>}\ E_1\ \mid\ x_2\ \texttt{=>}\ E_2{:}\tau}\text{typed-destruct}$$

$$\frac{\begin{array}{c}\Gamma(x) = \tau_c \qquad \tau_c \in \Xi \\ \tau \neq \langle \tau_d, \tau_c \rangle \qquad \Gamma + \{C \mapsto \langle \tau_d, \tau_c \rangle\},\Delta \vdash E{:}\tau\end{array}}{\Gamma,\Delta \vdash \texttt{let constructor}\ x\ \texttt{=}\ C\ \texttt{of}\ \tau_d\ \texttt{in}\ E\ \texttt{end}{:}\tau}\text{typed-dt-extend}$$

A.3 Dynamic Semantics

These are the dynamic semantics for mini-ML with extensible datatypes. We let E range over expressions, Γ over enviornments, τ over types, and x over variables in the language.

$$\frac{}{\Gamma \vdash \texttt{z} \hookrightarrow \texttt{z}}\text{ev-z} \qquad \frac{\Gamma \vdash E \hookrightarrow E}{\Gamma \vdash \texttt{s}\ \ E \hookrightarrow \texttt{s}\ \ E}\text{ev-s}$$

$$\frac{\Gamma \vdash E_1 \hookrightarrow \texttt{z} \qquad \Gamma \vdash E_2 \hookrightarrow V}{\Gamma \vdash \texttt{case}\ E_1\ \texttt{of z =>}\ E_2\ \mid\ \texttt{s}\ x\ \texttt{=>}\ E_3 \hookrightarrow V}\text{ev-case-z}$$

$$\frac{\Gamma \vdash E_1 \hookrightarrow \texttt{s}V_1 \qquad \Gamma + \{x \mapsto V_1\} \vdash E_3 \hookrightarrow V}{\Gamma \vdash \texttt{case}\ E_1\ \texttt{of z =>}\ E_2\ \mid\ \texttt{s}\ x\ \texttt{=>}\ E_3 \hookrightarrow V}\text{ev-case-s}$$

$$\frac{}{\Gamma \vdash \texttt{true} \hookrightarrow \texttt{true}}\text{ev-true} \qquad \frac{}{\Gamma \vdash \texttt{false} \hookrightarrow \texttt{false}}\text{ev-false}$$

$$\frac{\Gamma \vdash E_1 \hookrightarrow \texttt{false}}{\Gamma \vdash E_1\ \texttt{and}\ E_2 \hookrightarrow \texttt{false}}\text{ev-and-f\_} \qquad \frac{\Gamma \vdash E_2 \hookrightarrow \texttt{false}}{\Gamma \vdash E_1\ \texttt{and}\ E_2 \hookrightarrow \texttt{false}}\text{ev-and-\_f}$$

$$\frac{\Gamma \vdash E_2 \hookrightarrow \texttt{true} \qquad \Gamma \vdash E_2 \hookrightarrow \texttt{true}}{\Gamma \vdash E_1\ \texttt{and}\ E_2 \hookrightarrow \texttt{true}}\text{ev-and-\_f}$$

$$\frac{\Gamma \vdash E_1 \hookrightarrow \texttt{true}}{\Gamma \vdash E_1\ \texttt{or}\ E_2 \hookrightarrow \texttt{true}}\text{ev-or-t\_} \qquad \frac{\Gamma \vdash E_2 \hookrightarrow \texttt{true}}{\Gamma \vdash E_1\ \texttt{or}\ E_2 \hookrightarrow \texttt{true}}\text{ev-or-\_t}$$

$$\frac{\Gamma \vdash E_1 \hookrightarrow \texttt{false} \qquad \Gamma \vdash E_2 \hookrightarrow \texttt{false}}{\Gamma \vdash E_1 \ \texttt{or} \ E_2 \hookrightarrow \texttt{false}}\text{ev-or-ff}$$

$$\frac{\Gamma \vdash E \hookrightarrow \texttt{true}}{\Gamma \vdash \texttt{not} \ E \hookrightarrow \texttt{false}}\text{ev-not-f} \qquad \frac{\Gamma \vdash E \hookrightarrow \texttt{false}}{\Gamma \vdash \texttt{not} \ E \hookrightarrow \texttt{true}}\text{ev-not-t}$$

$$\frac{\Gamma \vdash E_{test} \hookrightarrow \texttt{true} \qquad \Gamma \vdash E_t \hookrightarrow V}{\Gamma \vdash \texttt{if} \ E_{test} \ \texttt{then} \ E_t \ \texttt{else} \ E_f \hookrightarrow V}\text{ev-if-true}$$

$$\frac{\Gamma \vdash E_{test} \hookrightarrow \texttt{false} \qquad \Gamma \vdash E_f \hookrightarrow V}{\Gamma \vdash \texttt{if} \ E_{test} \ \texttt{then} \ E_t \ \texttt{else} \ E_f \hookrightarrow V}\text{ev-if-false}$$

$$\frac{\Gamma \vdash E_1 \hookrightarrow V_1 \qquad \Gamma \vdash E_2 \hookrightarrow V_2}{\Gamma \vdash \texttt{(}E_1\texttt{,}E_2\texttt{)} \hookrightarrow \texttt{(}V_1\texttt{,}V_2\texttt{)}}\text{ev-pair}$$

$$\frac{\Gamma \vdash E \hookrightarrow \texttt{(}V_1\texttt{,}V_2\texttt{)}}{\Gamma \vdash \texttt{fst} \ E \hookrightarrow V_1}\text{ev-fst} \qquad \frac{\Gamma \vdash E \hookrightarrow \texttt{(}V_1\texttt{,}V_2\texttt{)}}{\Gamma \vdash \texttt{snd} \ E \hookrightarrow V_2}\text{ev-snd}$$

$$\frac{}{\Gamma \vdash \texttt{fn} \ x \ \texttt{=>} \ E \hookrightarrow \texttt{fn} \ x \ \texttt{=>} \ E}\text{ev-lam}$$

$$\frac{}{\Gamma \vdash \texttt{fn} \ x\texttt{:}\tau \ \texttt{=>} \ E \hookrightarrow \texttt{fn} \ x \ \texttt{=>} \ E}\text{ev-clam}$$

$$\frac{\Gamma \vdash E_1 \hookrightarrow \texttt{fn} \ x \ \texttt{=>} \ V_1 \qquad \Gamma \vdash E_2 \hookrightarrow V_2 \qquad \Gamma + \{x \mapsto V_2\} \vdash V_1 \hookrightarrow V}{\Gamma \vdash E_1 E_2 \hookrightarrow V}\text{ev-app}$$

$$\frac{\begin{array}{c}\Gamma \vdash E_1 \hookrightarrow \texttt{fn} \ x\texttt{:}\tau \ \texttt{=>} \ V_1 \qquad \Gamma \vdash E_2 \hookrightarrow V_2 \\ \Gamma + \{x \mapsto V_2\} \vdash V_1 \hookrightarrow V\end{array}}{\Gamma \vdash E_1 E_2 \hookrightarrow V}\text{ev-capp}$$

$$\frac{\Gamma \vdash E \ \texttt{(fix} \ E\texttt{)} \hookrightarrow V}{\Gamma \vdash \texttt{fix} \ E \hookrightarrow V}\text{ev-fix}$$

$$\frac{\Gamma \vdash \texttt{fix} \ E \hookrightarrow V}{\Gamma \vdash \texttt{fix} \ E\texttt{:}\tau \hookrightarrow V}\text{ev-cfix}$$

$$\frac{\Gamma \vdash E_1 \hookrightarrow V_1 \qquad \Gamma + \{x \mapsto V_1\} \vdash E_2 \hookrightarrow V}{\Gamma \vdash \texttt{let val} \ x\texttt{=}E_1 \ \texttt{in} \ E_2 \hookrightarrow V}\text{ev-let}$$

$$\frac{\Gamma \vdash \texttt{let val} \ x\texttt{=}E_1 \ \texttt{in} \ E_2 \hookrightarrow V}{\Gamma \vdash \texttt{let val} \ x\texttt{:}\tau\texttt{=}E_1 \ \texttt{in} \ E_2 \hookrightarrow V}\text{ev-clet}$$

$$\frac{\Gamma \vdash E \hookrightarrow V}{\Gamma \vdash E\texttt{:}\tau \hookrightarrow V}\text{ev-cexp}$$

$$\frac{\Gamma \vdash E \hookrightarrow V}{\Gamma \vdash \texttt{let type } x \texttt{ = } \tau \texttt{ in } E \hookrightarrow V}\text{ev-defty}$$

$$\frac{\Gamma \vdash E \hookrightarrow V}{\Gamma \vdash \texttt{let extensible datatype } x \texttt{ in } E \hookrightarrow V}\text{ev-defty}$$

$$\frac{\Gamma \vdash E \hookrightarrow V}{\Gamma \vdash C \;\; E \hookrightarrow C \;\; V}\text{ev-construct}$$

$$\frac{\Gamma \vdash E \hookrightarrow C \;\; V' \qquad \Gamma + \{x_1 \mapsto V'\} \vdash E_1 \hookrightarrow V}{\Gamma \vdash \texttt{destruct } E \texttt{ of } C \;\; x_1 \texttt{ => } E_1 \mid x_2 \texttt{ => } E_2 \hookrightarrow V}\text{ev-destruct-match}$$

$$\frac{\begin{array}{c}\Gamma \vdash E \hookrightarrow C' \;\; V' \qquad C' \neq C \\ \Gamma + \{x_2 \mapsto C' \;\; V'\} \vdash E_2 \hookrightarrow V\end{array}}{\Gamma \vdash \texttt{destruct } E \texttt{ of } C \;\; x_1 \texttt{ => } E_1 \mid x_2 \texttt{ => } E_2 \hookrightarrow V}\text{ev-destruct-mismatch}$$

$$\frac{\Gamma \vdash E \hookrightarrow V}{\Gamma \vdash \texttt{let constructor } x \texttt{ = } C \texttt{ of } \tau \texttt{ in } E \hookrightarrow V}\text{ev-dt-extend}$$

B Elf Semantics

B.1 utils

```
% strings of a type. ^ is cons and $ is the end.
string : type -> type.
^ : T -> string T -> string T. %infix right 8 ^
$ : string T.

letters : type.
a : letters. b : letters. c : letters.
r : letters. s : letters. t : letters. l : letters. n : letters. e : letters.
i : letters. o : letters. u : letters. y : letters.

% these are the names of the constructors, e.g. for this defintion,
% datatype 'a option = NONE | SOME of 'a
% the bits which would be of names are NONE and SOME.
names : type.
name : string letters -> names.

!= : names -> names -> type.  %infix left 10 !=

!=-sym : A != B -> B != A.
!=-trans : A != B -> B != C -> A != C.
!=-rec : name (A ^ $) != name (B ^ $) -> name B' !=  name A'
          -> name (A ^ A') != name (B ^ B').

% you need one rule for each letter -- just show that that
% letter is different from a.
```

```
!=-base-b : name (b ^ $) != name (a ^ $).
!=-base-c : name (c ^ $) != name (a ^ $).
!=-base-r : name (r ^ $) != name (a ^ $).
!=-base-s : name (s ^ $) != name (a ^ $).
!=-base-t : name (t ^ $) != name (a ^ $).
!=-base-l : name (l ^ $) != name (a ^ $).
!=-base-n : name (n ^ $) != name (a ^ $).
!=-base-e : name (e ^ $) != name (a ^ $).
!=-base-i : name (i ^ $) != name (a ^ $).
!=-base-o : name (o ^ $) != name (a ^ $).
!=-base-u : name (u ^ $) != name (a ^ $).
!=-base-y : name (y ^ $) != name (a ^ $).

unary : type.
0 : unary.

% natual numbers -- a unary string.
nats : type. %name nats N.
nat : string unary -> nats. %prefix 7 nat

% arities
arities : type. %name arities A
arity: nats -> arities.
```

B.2 types

```
%% kinds for mini-ml.
kind : type. %name kind K.
t : kind.
==> : kind -> kind -> kind. %infix right 8 ==>
tuple : arities -> kind.

%% read this as "is a type of this kind"
cn : kind -> type. %name cn T

%% basic type definition for mini-ml
ty-nat : cn t.
ty-bool : cn t.

% cross product
* : cn t -> cn t -> cn t. %infix left 8 *

% arrow types
--> : cn t -> cn t -> cn t. %infix right 9 -->

% type sequences.
ty-seq : arities -> type. %name ty-seq TS
ty-seq-nil : ty-seq (arity (nat $)).
ty-seq-cons : cn t -> ty-seq (arity (nat N)) ->
                      ty-seq (arity (nat 0 ^ N)).

% a useful thingy on type sequences.
```

```
%     NB: these are one based, not zero based -- there is no zero-th
%         index in the type sequences.
proj : ty-seq A -> nats -> cn t -> type.
proj-seq-base : proj (ty-seq-cons CN TS) (nat 0 ^ $) CN.
proj-seq-rec : proj TS (nat N) CN -> proj (ty-seq-cons CN' TS) (nat 0 ^ $) CN.

% constructors are represented by their name, the type of the domain,
% and the type of the range.
constructors : type. %name constructors C
constructor : names     % the name of the constructor
               -> cn t  % the type of the domain
               -> cn t  % the datatype for which this is a constructor
               -> constructors.

% object language type definition

% ty-inst is "type t = ..." in sml and here becuase of the higher-order
% abstract syntax is very boring.
ty-inst : cn t -> cn t.

% dt-inst corresponds to "extensible datatype t".
dt-inst : cn t.
```

B.3 expressions

```
%% expression definition for mini-ml
exp  : type. %name exp E

% natural numbers
z    : exp.
s    : exp -> exp.
case : exp -> exp -> (exp -> exp) -> exp.

% booleans
true : exp.
false : exp.
and : exp -> exp -> exp.  %infix left 8 and
or : exp -> exp -> exp.   %infix left 8 or
not : exp ->  exp.        %prefix 9 not
if : exp -> exp -> exp -> exp.

% pairs
, : exp -> exp -> exp. %infix left 10 ,
fst  : exp -> exp.
snd  : exp -> exp.

lam  : (exp -> exp) -> exp.
app  : exp -> exp -> exp.
let  : exp -> (exp -> exp) -> exp.
fix  : (exp -> exp) -> exp.

cexp : exp -> cn t -> exp.
```

```
% defining types and datatypes.
defty : cn t -> (cn t -> exp) -> exp.

% the type specified must be the domain of the function.
clam : cn t -> (exp -> exp) -> exp.

% the type specified must be the domain of the function.
cfix : cn t -> (exp -> exp) -> exp.

% the type specified is the type of the bound variable.
clet : exp -> cn t -> (exp -> exp) -> exp.

% This is how you build objects that have the type of a datatype.
construct : constructors -> exp -> exp.

% This is how you destruct objects of a datatype.
destruct : constructors     % the constructor to test for
           -> exp           % the expression being tested
           -> (exp -> exp)  % matching case -- in this case the bound exp
                            % will be what's "inside" counstructor
           -> (exp -> exp)  % didn't match case -- the bound exp will
                            % be the same as the expression being tested.
           -> exp.

% These are the extensible datatypes.
dt-extend : constructors -> (constructors -> exp) -> exp.
```

B.4 static

```
%% static typing of mini-ml
typed : exp -> cn t -> type. %name typed U

typed-z : typed z ty-nat.
typed-s : typed E ty-nat -> typed (s E) ty-nat.
typed-case : typed E1 ty-nat -> typed E2 T ->
             typed (lam E3) (ty-nat --> T) -> typed (case E1 E2 E3) T.

typed-true : typed true ty-bool.
typed-false : typed false ty-bool.
typed-and : typed E1 ty-bool -> typed E2 ty-bool -> typed (E1 and E2) ty-bool.
typed-or : typed E1 ty-bool -> typed E2 ty-bool -> typed (E1 or E2) ty-bool.
typed-not : typed E ty-bool -> typed (not E) ty-bool.
typed-if : typed TEST ty-bool -> typed E1 T -> typed E2 T
                              -> typed (if TEST E1 E2) T.

typed-pair : typed E1 T1 -> typed E2 T2 ->
             typed (E1 , E2) (T1 * T2).
typed-fst : typed E (T1 * T2) -> typed (fst E) T1.
typed-snd : typed E (T1 * T2) -> typed (snd E) T2.

typed-lam : ({E1:exp} typed (E2 E1) Tc -> typed E1 Td) ->
            typed (lam E2) (Td --> Tc).
typed-clam : ({E1:exp} typed (E2 E1) Tc -> typed E1 Td) ->
```

```
             typed (clam Td E2) (Td --> Tc).
typed-app : typed E1 (Td --> Tc) -> typed E2 Td -> typed (app E1 E2) Tc.
typed-fix : typed (lam E) (T --> T) -> typed (fix E) T.
typed-cfix : typed (fix E) (T --> T) -> typed (cfix T E) T.

typed-cexp : typed E T -> typed (cexp E T) T.

% This is how I first typed let expressions. However, to aid in the
% elf representation of the representation I changed it to the
% subsequent definition.

%typed-let : typed E1 Td -> typed (lam E2) (Td --> Tc) ->
%           typed (let E1 E2) Tc.

typed-let : typed E1 Td ->
           ({x:exp} ({T:cn t} typed E1 T -> typed x T ) -> typed (E2 x) Tc) ->
           typed (let E1 E2) Tc.

typed-clet : typed E1 Td ->
           ({x:exp} ({T:cn t} typed E1 T -> typed x T ) -> typed (E2 x) Tc) ->
           typed (clet E1 Td E2) Tc.

% This is the case where you are defining a regular type
typed-defty-ty : typed (E (ty-inst T')) T -> typed (defty (ty-inst T') E) T.

% this is the case where you are defining a datatype.
typed-defty-dt : typed (E dt-inst) T -> typed (defty dt-inst E) T.

typed-construct : typed E Td -> typed (construct (constructor N Td Tc) E) Tc.

typed-destruct : typed TEST Tc ->
  ({x:exp} ({T':cn t} typed E1 T' -> typed x T') -> typed (MISMATCH x) T) ->
  ({x:exp} typed x Td -> typed (MATCH x) T) ->
  typed (destruct (constructor N Td Tc) TEST MATCH MISMATCH) T.

typed-dt-extend : typed (CE C) T -> typed (dt-extend C CE) T.
```

B.5 dynamic

```
% evaluator for mini-ml
eval : exp -> exp -> type. %name eval D
ev-z : eval z z.
ev-s : eval E V -> eval (s E) (s V).
ev-case-z : eval E1 z -> eval E2 V -> eval (case E1 E2 E3) V.
ev-case-s : eval E1 (s V1) -> eval (E3 V1) V2 -> eval (case E1 E2 E3) V2.

ev-true : eval true true.
ev-false : eval false false.
ev-and-f_ : eval E1 false -> eval (E1 and E2) false.
ev-and-_f : eval E2 false -> eval (E1 and E2) false.
ev-and-tt : eval E1 true -> eval E2 true -> eval (E1 and E2) true.
ev-or-t_ : eval E1 true -> eval (E1 or E2) true.
```

```
ev-or-_t : eval E2 true -> eval (E1 or E2) true.
ev-or-ff : eval E1 false -> eval E2 false -> eval (E1 or E2) false.
ev-not-t : eval E false -> eval (not E) true.
ev-not-f : eval E true -> eval (not E) false.
ev-if-true : eval TEST true -> eval T V -> eval (if TEST T F) V.
ev-if-false : eval TEST false -> eval F V -> eval (if TEST T F) V.

ev-pair : eval E1 V1 -> eval E2 V2 -> eval (E1 , E2) (V2 , V2).
ev-fst : eval E (E1 , E2) -> eval (fst E) E1.
ev-snd : eval E (E1 , E2) -> eval (snd E) E2.

ev-lam : eval (lam E) (lam E).
ev-clam : eval (clam T E) (clam T E).

ev-fix : eval (E (fix E)) V -> eval (fix E) V.
ev-cfix : eval (fix E) V -> eval (cfix T E) V.

ev-app : eval E1 (lam V1) -> eval E2 V2 ->
         eval (V1 V2) V -> eval (app E1 E2) V.
ev-capp : eval E1 (clam T V1) -> eval E2 V2 -> eval (V1 V2) V ->
          eval (app E1 E2) V.

ev-let : eval E1 V1 -> eval (E2 V1) V -> eval (let E1 E2) V.
ev-clet : eval E1 V1 -> eval (E2 V1) V -> eval (clet E1 T E2) V.

ev-cexp : eval E V -> eval (cexp E TE) V.

ev-defty : eval (E T) V -> eval (defty T E) V.

ev-construct : eval E V -> eval (construct C E) (construct C V).
ev-destruct-match : eval TEST (construct C TEST-V)
                     -> eval (MATCH TEST-V) V
                     -> eval (destruct C TEST MATCH MISMATCH) V.

ev-destruct-mismatch :
      eval TEST (construct (constructor N T T') TEST-V)
   -> N != N'
   -> eval (MISMATCH (construct (constructor N T T') TEST-V)) V
   -> eval (destruct (constructor N' T'' T''') TEST MATCH MISMATCH) V.

ev-dt-extend : eval (CE C) V -> eval (dt-extend C CE) V.
```

B.6 value

```
%Value definition for mini-ml
value : exp -> type. %name value P
val-z : value z.
val-s : value V -> value (s V).
val-true : value true.
val-false : value false.
val-pair : value V1 -> value V2 -> value (V1 , V2).
val-lam : value (lam V).
val-clam : value (clam T V).
```

```
val-construct : value V -> value (construct C V).

% value soundness for mini-ml
vs : eval E V -> value V -> type.
vs-z : vs ev-z val-z.
vs-s : vs D P -> vs (ev-s D) (val-s P).
vs-case-z : vs D2 P -> vs (ev-case-z D1 D2) P.
vs-case-s : vs D2 P -> vs (ev-case-s D1 D2) P.

vs-true : vs ev-true val-true.
vs-false : vs ev-false val-false.
vs-and-f_ : vs D val-false -> vs (ev-and-f_ D) val-false.
vs-and-_f : vs D val-false -> vs (ev-and-_f D) val-false.
vs-and-tt : vs D1 val-true -> vs D2 val-true -> vs (ev-and-tt D1 D2) val-true.
vs-or-t_ : vs D val-true -> vs (ev-or-t_ D) val-true.
vs-or-_t : vs D val-true -> vs (ev-or-_t D) val-true.
vs-or-ff : vs D1 val-false -> vs D2 val-false ->
           vs (ev-or-ff D1 D2) val-false.
vs-not-t : vs D val-false -> vs (ev-not-t D) val-true.
vs-not-f : vs D val-true -> vs (ev-not-f D) val-false.
vs-if-true : vs DT val-true -> vs D P -> vs (ev-if-true DT D) P.
vs-if-true : vs DT val-false -> vs D P -> vs (ev-if-false DT D) P.

vs-pair : vs D1 P1 -> vs D2 P2 -> vs (ev-pair D1 D2) (val-pair P1 P2).
vs-fst : vs D (val-pair P1 P2) -> vs (ev-fst D) P1.
vs-snd : vs D (val-pair P1 P2) -> vs (ev-fst D) P2.
vs-lam : vs ev-lam val-lam.
vs-app : vs D3 P3 -> vs (ev-app D1 D2 D3) P.
vs-let : vs D2 P -> vs (ev-let D1 D2) P.
vs-fix : vs D1 P -> vs (ev-fix D1) P.
vs-cexp : vs D P -> vs (ev-cexp D) P.
vs-clam : vs ev-clam val-clam.
vs-clet : vs D2 P -> vs (ev-clet D1 D2) P.

% wow -- somehow when elf expands this, the constructor inside
% ev-construct is required to be the same as the constructor inside
% val-construct.  how did that happen?

vs-construct : vs D P -> vs (ev-construct D) (val-construct P).
vs-destruct-match :  vs D2 P -> vs (ev-destruct-match D1 D2) P.
vs-destruct-mismatch : vs D2 P ->
                       vs (ev-destruct-mismatch D1 NOT-EQUAL D2) P.
vs-dt-extend : vs D P -> vs (ev-dt-extend D) P.
```

B.7 type-soundness

```
%% Proof of type soundness for mini-ml
ts : eval E Y -> typed E T -> typed Y T -> type.

%% This is a special case of the replacement lemma
%% used for the proof of type soundness.
% The cases which define it are in replace.elf
```

```
replace : ({x:exp} ({T:cn t} typed E1 T -> typed x T) -> typed (E2 x) T2) ->
           eval E1 V1 -> typed (E2 V1) T2 -> type.

ts-z : ts ev-z typed-z typed-z.
ts-s : ts D X Y -> ts (ev-s D) (typed-s X) (typed-s Y).

ts-true : ts ev-true typed-true typed-true.
ts-false : ts ev-false typed-false typed-false.
ts-and-f_ : ts (ev-and-f_ D) (typed-and X X') typed-false.
ts-and-_f : ts (ev-and-_f D) (typed-and X X') typed-false.
ts-and-tt : ts (ev-and-tt D D') (typed-and X X') typed-true.
ts-or-t_ : ts (ev-or-t_ D) (typed-or X X') typed-true.
ts-or-_t : ts (ev-or-_t D) (typed-or X X') typed-true.
ts-or-ff : ts (ev-or-ff D D') (typed-or X X') typed-false.
ts-not-t : ts (ev-not-t D) (typed-not X) typed-true.
ts-not-f : ts (ev-not-f D) (typed-not X) typed-false.
ts-if-true : ts D2 X2 Y ->
             ts (ev-if-true D1 D2) (typed-if X1 X2 X3) Y.
ts-if-false : ts D2 X3 Y ->
              ts (ev-if-false D1 D2) (typed-if X1 X2 X3) Y.

ts-pair : ts D1 X1 Y1 -> ts D2 X2 Y2 ->
          ts (ev-pair D1 D2) (typed-pair X1 X2) (typed-pair Y1 Y2).
ts-fst : ts D X (typed-pair Y Y') -> ts (ev-fst D) (typed-fst X) Y.
ts-snd : ts D X (typed-pair Y' Y) -> ts (ev-snd D) (typed-snd X) Y.

ts-lam : ts ev-lam (typed-lam X) (typed-lam X).
ts-clam : ts ev-clam (typed-clam X) (typed-clam X).

ts-app : ts D1 X1 (typed-lam Y1) -> ts D2 X2 Y2 -> ts D3 (Y1 Y3 Y2) Y ->
         ts (ev-app D1 D2 D3) (typed-app X1 X2) Y.

ts-fix : ts (ev-fix D) (typed-fix X) (typed-fix X).
ts-cfix : ts (ev-cfix D) (typed-cfix X) (typed-cfix X).

ts-let : ts D2 X2' Y -> replace X2 D1 X2' ->
         ts (ev-let D1 D2) (typed-let X1 X2) Y.

ts-clet : ts D2 X2' Y -> replace X2 D1 X2' ->
          ts (ev-clet D1 D2) (typed-clet X1 X2) Y.

ts-cexp : ts (ev-cexp D) (typed-cexp X) (typed-cexp X).

ts-defty-dt : ts D X Y ->
              ts (ev-defty D) (typed-defty-ty X) (typed-defty-ty Y).
ts-defty-ty : ts D X Y ->
              ts (ev-defty D) (typed-defty-dt X) (typed-defty-ty Y).

ts-destruct-match : ts D1 X1 (typed-construct Y1) ->
                    ts D2 (X3 Y2 Y1) Y ->
                    ts (ev-destruct-match D1 D2) (typed-destruct X1 X2 X3) Y.

ts-destruct-mismatch : ts D3 X2' Y -> replace X2 D1 X2' ->
```

```
                        ts (ev-destruct-mismatch D1 D2 D3)
                           (typed-destruct X1 X2 X3) Y.

ts-construct : ts D X Y ->
               ts (ev-construct D) (typed-construct X) (typed-construct Y).
ts-dt-extend : ts D X Y ->
               ts (ev-dt-extend D) (typed-dt-extend X) (typed-dt-extend Y).
```

B.8 replace

```
% This is a special case of the replacement lemma
% used for the proof of type soundness.

% This is defined in type-soundness.
%replace : ({x:exp} ({T:cn t} typed E1 T -> typed x T) -> typed (E2 x) T2) ->
%           eval E1 V1 -> typed (E2 V1) T2 -> type.

rp_hyp : ts D X Y ->
         replace ([x:exp] [u:{T':cn t} typed E1 T' -> typed x T'] u T X) D Y.
```

References

[BHA86] G.L. Burn, C. Hankin, and S. Abramsky. Stricness analysis for higher-order functions. *Science of Computer Programming*, 7(3):249–78, November 1986.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximations of fixpoints. In *Fourth Annual ACM Symposium on Principles of Programming Languages*, 1977.

[Col] Christopher P. Colby. Computing invariants via abstract interpretation. internal correspondence.

[Con90] Charles Consel. Binding time analysis for higher order untyped functional languages. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 264–272, 1990.

[Deu92] Alain Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *Proceedings of the IEEE 1992 International Conference on Computer Languages*, San Fransisco, California, pages 2–13, April 1992.

[FSDF93] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. *Conference on Programming Language Design and Imlementation*, June 1993.

[GLS94] Jr. Guy L. Steele. Building interpreters by composing monads. In *Proceedings of the Symposium of Principles of Programming Languages*, January 1994.

[Hec77] M. S. Hecht. Flow analysis cof computer programs. North-Holland, New-York, 1977.

[JGB+90] Neil D. Jones, Carsten K. Gomard, Anders Bondorf, Olivier Danvy, and Torben Mogensen. A self-applicable partial evaluator for the lambda calculus. In *Proceedings of the IEEE International Conference on Computer Languages*, pages 49–58, 1990.

[Jon94] Mark P. Jones. The implementation of the gofer functional programming system. Technical Report YALEU/DCS/RR-1030, Yale University Department of Computer Science, May 1994.

[LHJ95] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. *Principles of Programming Languages*, 1995.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal or Computer and System Science*, 17:348–375, 1978.

[Mog91a] Eugenio Moggi. A modular approach to denotational semantics. In D. H. Pitt, P.-L. Curien, S. Abramsky, A. M. Pitts, A. Poigne, and D. E. Rydeheard, editors, *Category Theory and Computer Science, Lecture Notes in Computer Science*, volume 301. Springer-Verlag, 1991.

[Mog91b] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, July 1991.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[Pfe91] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[Rep91] J. H. Reppy. CML: a higher-order concurrent language. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, 1991.

[Wad92] Philip Wadler. The essence of functional programming. In *Proceedings of the 19'th Annual Symposium on Principles of Programming Languages*, 1992.