Incorporating Static Analysis in a Combinator-Based Compiler*

MARGARET MONTENYOHUL

The Center for Advanced Computer Studies, University of Southwestern Louisiana, Lafayette, Louisiana 70504

AND

MITCHELL WAND

Northeastern University, Boston, Massachusetts 02115

We show how restructuring a denotational definition leads to a more efficient compiling algorithm. Three semantics-preserving transformations (static replacement, factoring, and combinator selection) are used to convert a continuation semantics into a formal description of a semantic analyzer and code generator. The compiling algorithm derived below performs type checking before code generation so that type-checking instructions may be omitted from the target code. The optimized code is proved correct with respect to the original definition of the source language. The proof consists of showing that all transformations preserve the semantics of the source language.

INTRODUCTION

Several researchers have investigated methodologies for deriving an implementation from a language's denotational definition (Hudak and Kranz 1983; Jones and Muchnick, 1982; Nielson, 1985; Pleban, 1981; Turner, 1979; Wand, 1980, 1982a, 1982b, 1983). The primary goal in this area is to develop an efficient implementation from a formal definition in a such a way that correctness can be verified. Typically, a standard denotational definition specifies the run-time behavior of a construct. Such a definition may be translated easily into an interpreter and, although it may be inefficient, the implementation is guaranteed to be correct. The methodology presented in (Wand, 1980, 1982a, 1982b, 1983) compiles a program by applying meaning-preserving transformations to a language's continuation semantics. The correctness proof is straightforward; it

* This work supported by Grant NSF-MCS 82-03978 and NSF-DCR 83-03325.
depends on showing a direct relationship between the functions specifying
the meaning of each construct and the representation chosen for each
construct. In this paper, this methodology is extended to derive a compiler
which uses data flow analysis to produce more efficient target code. The
extended methodology can be described as a four step process:

(1) Write a standard continuation semantics for the source language.

(2) Identify computations which may be performed before program
execution and transform those computations into static operations (static
replacement).

(3) Rearrange the definition so that the static operations are
evaluated before and independently of dynamic interpretation (factoring).

(4) Choose a representation for the dynamic meaning of each
construct in the language (combinator selection).

Steps one through four are illustrated below for an expression language
called EL. First, a standard dynamic semantics is written. EL is strongly
typed therefore type-checking operations may be expressed as static com-
putations. The static replacement step converts run-time type checking
operations appearing in the dynamic definition into equivalent static
computations. Factoring rearranges the semantic clauses so that static type
checking is performed independently of and prior to dynamic inter-
pretation. The resulting definition is suitable for compiler derivation using
the methodology already described in (Wand, 1980, 1982a, 1982b, 1983)
which consists of three steps. First, a continuation semantics is written in
combinator terms (combinator selection). Second, associative properties of
the combinators are employed to rotate terms into a linear form. Third,
machine actions, which reduce the terms to normal form, are derived from
the semantic clauses.

The purpose of this work was to investigate how program improvement
techniques used by realistic compilers could be rigorously described in a
denotational framework. In particular, how could a formal description of
data flow analysis be used to yield a better compiler than the sort derived
in (Wand, 1980, 1982a, 1982b, 1983)? We found writing a denotational
description of the type constraints of EL to be a straightforward process.
The static definition closely models typical compiler behavior; constructs
are analyzed for type information using a static environment in much the
same way that a compiler performs type checking using a symbol table
(compile-time environment). Incorporating static analysis into a dynamic
definition results in a formal specification which expresses type checking in
terms of compile-time information. Target code derived from such a
modified definition is more efficient than code derived directly from the
original dynamic definition. Although this paper only discusses type
checking, this methodology is applicable to other program improvements (Montenyohl, 1986).

1. Static Replacement

Table I contains the syntax and continuation semantics of EL. The auxiliary functions (terminate, etc.) and error messages are displayed in the Appendix. The value of an expression can have one of three types: integer, real, or boolean. Therefore, the domain of expressed values is the disjoint union of three domains: Int, Real, and Bool. Type checking can be expressed using the domain operation for inspection. For instance, the type of an expressed Eval(e (v)) can be determined by evaluating (isInt? v), (isReal? v) or (isBool? v). The conditional expressions are written using a guarded command style rather than deeply nested \( \Rightarrow \), phrases. The domain inspection predicates represent run-time type-checking operations because \( v \) is a run-time value.

### TABLE I

**Definition of EL**

<table>
<thead>
<tr>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Exp) ::= (Const)</td>
</tr>
<tr>
<td>(Const) ::= int (Intconst)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dynamic Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eval : Int + Real + Bool</td>
</tr>
<tr>
<td>Econt : Eval \rightarrow Answer</td>
</tr>
<tr>
<td>Dval : Eval + ('undeclared')</td>
</tr>
<tr>
<td>Env : Ide \rightarrow Dval</td>
</tr>
<tr>
<td>( \mathcal{E} )</td>
</tr>
</tbody>
</table>

\[
\mathcal{E}[\text{int } c] = \text{int-const } c \\
\mathcal{E}[\text{real } c] = \text{real-const } c \\
\mathcal{E}[\text{bool } c] = \text{bool-const } c \\
\mathcal{E}[i] = \lambda \text{pc. } \mu = \text{undeclared} \Rightarrow \text{terminate error} \,
\text{Eval(e)} \\
\mathcal{E}[\text{let } c_1 = c_2 \text{ in } c_3] = \lambda \text{pc. } \mathcal{E}[c_3](\text{eval } \text{pc}(\text{eval } \text{pc}(c_2, \text{eval } \text{pc}(c_1)))) \\
\mathcal{E}[c_1 + c_2] = \lambda \text{pc. } \mathcal{E}[c_1](\text{eval } \text{pc}(\text{eval } \text{pc}(\text{eval } \text{pc}(c_2))) \text{eval } \text{pc}(\text{eval } \text{pc}(c_1))) \\
\text{isInt? } v_1 : \\
\mathcal{E}[v_2]p(\lambda \text{v}_2). \\
\text{isInt? } v_2 : \mathcal{E}[(v_1 \mid \text{Int} + (v_2 \mid \text{Int})) \text{eval } \text{pc}]
\text{isReal? } v_2 : \mathcal{E}[(v_1 \mid \text{Real} + (v_2 \mid \text{Real})) \text{eval } \text{pc}]
\text{isBool? } v_2 : \mathcal{E}[(v_1 \mid \text{Bool} + (v_2 \mid \text{Bool})) \text{eval } \text{pc}]
\text{terminate error} \\
\text{isReal? } v_2 : \mathcal{E}[(v_1 \mid \text{Real} + (v_2 \mid \text{Real})) \text{eval } \text{pc}]
\text{isBool? } v_2 : \mathcal{E}[(v_1 \mid \text{Bool} + (v_2 \mid \text{Bool})) \text{eval } \text{pc}]
\text{terminate error} \\
\text{isBool? } v_2 : \mathcal{E}[(v_1 \mid \text{Bool} + (v_2 \mid \text{Bool})) \text{eval } \text{pc}]
\text{terminate error}
If a language is strongly typed, one may define type-checking computations as functions of syntax, a set of types, and an environment that maps identifiers to types. Table II contains a direct denotational definition of the type constraints of EL. The function $\delta_i$ takes an expression and a type environment and returns the type of the expression. If a type error exists in the expression, it returns "untyped."

Note the plus sign used in the denotation for an addition expression is overloaded; in the expression $((u_1 \text{ Real}) + (u_2 \text{ Real}))$, the plus sign denotes real addition whereas the plus sign in the expression $((0, 1 \text{ Bool}) + (u_2 \text{ Bool}))$ denotes logical disjunction.

The static definition does not mention coercions explicitly, but the static value of a construct depends on the coercion rules for the language. For example, if $x$ is bound to "real" and $y$ is bound to "int," then the value of $\delta_i((x + y)) \tau$ is "real." Only integer-to-real coercion is allowed in EL.

Theorem 1.2, given below, asserts that run-time analysis in the dynamic definition of EL may be replaced by static analysis with no loss of meaning. In order to prove this theorem, it is necessary to establish a relationship between the static environment and the dynamic environment. The Type Consistency Property shown below states this relationship. It asserts, in the first item, that the dynamic environment is defined for all identifiers; applying the environment to an identifier yields either "undeclared" or a
non-bottom element of \text{Eval}. The remaining items state that the static environment must correspond to the dynamic environment in two ways: (ii) if an identifier is undeclared during execution then it is also statically bound to "untyped"; (iii) if an identifier is bound at run-time to an element of \text{Int}, \text{Real}, or \text{Bool} then the static environment binds that identifier to the type "int," "real," or "bool," respectively.

Type Consistency Property. Let \( \tau \in \text{Type-Env}, \rho \in \text{Env}; \langle \tau, \rho \rangle \) are type consistent if and only if for all \( i \in \text{Ide}, \)

\[
\begin{align*}
\text{(i)} & \quad \rho_i \neq \perp_{\text{Dvalue}} \text{ and } [\rho_i \mid \text{Eval}] \neq \perp_{\text{Eval}} \\
\text{(ii)} & \quad \tau_i = "\text{untyped}" \iff \rho_i = "\text{undeclared}" \\
\text{(iii)} & \quad \tau_i = "\text{int}" \iff \text{isInt?} (\rho_i), \\
& \quad \tau_i = "\text{real}" \iff \text{isReal?} (\rho_i), \text{ and} \\
& \quad \tau_i = "\text{bool}" \iff \text{isBool?} (\rho_i).
\end{align*}
\]

Consistency is a reasonable property to have between the environments since the static binding of an identifier is derived by analyzing the same declaration that is used to determine the run-time binding; the declaration is used in static analysis to determine a type and it is used at run-time to determine a value for the identifier. The environment grows during expression evaluation. Lemma 1.1, stated below, asserts that the operation used to extend each environment preserves type consistency. A bit of notation is introduced to facilitate the statement of Lemma 1.1: the expression \( \text{consis?} \langle \tau, \rho \rangle \) is true if \( \langle \tau, \rho \rangle \) are type consistent. The proof is done by structural induction on \( e \), as shown in the Appendix.

Theorem 1.2 states the correctness of replacing a run-time test of an expressible value with the static test of the expression. Knowing that a value is associated with the domain \text{Int}, \text{Real}, or \text{Bool} is equivalent to knowing that the static value of the expression is "int," "real," or "bool," respectively. The proof is done by structural induction on \( e \). The interesting cases, the let and the addition expression, are shown in the Appendix.

\begin{lemma}
Let \( \langle \tau, \rho \rangle \) be type consistent. Let \( \epsilon \in \text{Answer} \). For all \( \tau \in \text{Type-Env}, \rho \in \text{Dynamic-Env}, \) let \( \text{consis?} \langle \tau, \rho \rangle \iff \langle \tau, \rho \rangle \) are type consistent. Let \( x \in \text{Ide} \) be arbitrary but fixed.

For all \( e \in \text{Exp} \), and \( \epsilon \in \text{Econt} \), if \( \delta[[e]] \tau \neq "\text{untyped}" \) then

\[
\begin{align*}
\delta[[e]] \rho (\lambda v. \text{consis?} ((\text{ext-tau} x\tau[\delta[[e]] \tau]), \\
(\text{ext-rho} x\rho v)) \Rightarrow \epsilon v, \text{error} = \delta[[e]] \rho \epsilon.
\end{align*}
\]
\end{lemma}

\begin{theorem} [static replacement]
Let \( \langle \tau, \rho \rangle \) be type consistent. For all \( e \in \text{Exp} \) and for all \( f \in \text{Econt}, 1 \leq j \leq 4, \)
\end{theorem}
if $\delta_i[e]\tau \neq \text{"untyped"}$ then

\[\delta_i[e] \rho(\lambda v \cdot \text{isInt? } v : f_1 v)\]
\[= \delta_i[e] \rho(\lambda v \cdot \delta_i[e] \tau = \text{"int" } \Rightarrow f_1 v)\]
\[\delta_i[e] \tau = \text{"real" } \Rightarrow f_2 v\]
\[\delta_i[e] \tau = \text{"bool" } \Rightarrow f_3 v\]
\[\delta_i[e] \tau = \text{"int" } \Rightarrow f_4 v\]

2. FACTORING

After dynamic type checking is replaced by static analysis, then static analysis may be factored out of the dynamic clauses. The corollary below rearranges semantic clauses so that the static meaning of a construct is separated from its dynamic meaning. Lemma 2.1, stated below, cites a general result for the lambda calculus and the corollary that follows addresses this result to the definition of EL. The lemma states that the boolean expression of a conditional expression may be moved outside the lambda binding if the identifiers in the boolean expression are not bound by the lambda. Corollary 2.2, a trivial consequence of Lemma 2.1, is used to factor the dynamic meaning of EL. In the factored definition, the type of a syntactic object is analyzed before the object is evaluated by the dynamic semantic functions. The factorization hinges on the fact that the static analysis does not depend on any run-time values. The case notation in the corollary is more convenient than the arrow notation for writing conditional clauses for EL.

**LEMMA 2.1.** Let $b \in \text{Bool}$ and $b \neq \bot$. For all $m \in \text{Econt} \rightarrow \text{Answer}$ and $p, q \in \text{Econt},$

\[m(\lambda v \cdot b \Rightarrow pv, qv) = b \Rightarrow mp, mq.\]

**COROLLARY 2.2 (factorization).** Let $\rho \in \text{Env}$, $e \in \text{Econt}$, and $\tau \in \text{TypeEnv}$. For all $e \in \text{Exp}$, and $g_j \in \text{Econt}, 1 \leq j \leq 4,$

\[\delta_i[e] \rho(\lambda v \cdot \delta_i[e] \tau = \text{"int" } \Rightarrow g_1 v,\]
\[\delta_i[e] \tau = \text{"real" } \Rightarrow g_2 v,\]
\[\delta_i[e] \tau = \text{"bool" } \Rightarrow g_3 v, g_4 v)\]
\[=\]
\[\delta_i[e] \tau = \text{"int" } \Rightarrow \delta_i[e] \rho g_1,\]
\[\delta_i[e] \tau = \text{"real" } \Rightarrow \delta_i[e] \rho g_2,\]
\[\delta_i[e] \tau = \text{"bool" } \Rightarrow \delta_i[e] \rho g_3, \delta_i[e] \rho g_4.\]
= case $\delta_\tau[e]$ of
  "int" : $\delta[e] \rho g_1$
  "real" : $\delta[e] \rho g_2$
  "bool" : $\delta[e] \rho g_3$
  "untyped" : $\delta[e] \rho g_4$

A version of the definition for EL, presented in Table III, expresses type checking in terms of syntax and a type environment. Type errors are identified, coercions are inserted, and primitive operations are selected on the basis of the values of the static functions defined in Table II. The new definition is called the factored definition because the static meaning is factored from the dynamic meaning. The new semantic function takes an expression and a type environment as its first two arguments. For a well-typed expression, it returns a function which described the run-time evaluation of the expression. If the expression is not well typed, it returns the function \textit{terminate} with an error message.

\textbf{TABLE III}

Factored Semantics

\begin{verbatim}
E' : Exp → Type-Env → Env → Econt → Answer
E'[int c]τ = int-const c
E'[real c]τ = real-const c
E'[bool c]τ = bool-const c
E'[e]τ = λρ. case E[e] of
        'untyped' : terminate error,
        else : e (E[ρ]) [Eval]
E'[let * - ej in es,]τ =
        λρ. case E[e] of
        'untyped' : terminate error,
        else : case E[e] of
                           'int' : E[e]τ (λυ₁. E'[ej]τ (λυ₂, int-odd υ₁υ₂)),
                           'real' : E[e]τ (λυ₁, E'[ej]τ (λυ₂, real-odd υ₂υ₁)),
                           'bool' : E[e]τ (λυ₁, E'[ej]τ (λυ₂, terminate error)),
                           'untyped' : E[e]τ (λυ₁, E'[ej]τ (λυ₂, terminate error)),
        else : case E[e] of
                           'int' : E[e]τ (λυ₁, E'[ej]τ (λυ₂, terminate error)),
                           'real' : E[e]τ (λυ₁, E'[ej]τ (λυ₂, real-add υ₁υ₂)),
                           'bool' : E[e]τ (λυ₁, E'[ej]τ (λυ₂, terminate error)),
                           'untyped' : E[e]τ (λυ₁, E'[ej]τ (λυ₂, terminate error)),
        'bool':
        case E[e] of
        'int' : E[e]τ (λυ₁, E'[ej]τ (λυ₂, terminate error)),
        'real' : E[e]τ (λυ₁, E'[ej]τ (λυ₂, terminate error)),
        'bool' : E[e]τ (λυ₁, E'[ej]τ (λυ₂, terminate error)),
        'untyped' : E[e]τ (λυ₁, E'[ej]τ (λυ₂, terminate error)),
\end{verbatim}
For every EL construct, the factored clause gives the same semantics as the dynamic definition provided there are no untypable objects in the expression. Theorem 2.3 formalizes this assertion. The proof, shown in the Appendix, is done by structural induction on expressions.

**Theorem 2.3.** Let $\langle \tau, \rho \rangle$ be type consistent. Let $\delta$ be the dynamic function defined in Table I. Let $\delta'_s$ be the static function defined in Table II. Let $\delta'_t$ be the factored function defined in Table III. For all $e \in \text{Expression}$ and $e \in \text{Econt}$,

$$\text{if } \delta'_s[e] \tau \neq \text{"untyped" then } \delta'_t[e] \rho e = \delta'_t[e] \tau \rho e.$$ 

### 3. ADVANTAGES OF FACTORING

Factoring separates type analysis from the run-time interpretation of each construct. This allows for better code generation. In this section we review the code generation methodology presented in (Wand, 1980, 1982a, 1982b, 1983) and show how factoring improves the compiler’s output.

Each lambda expression in Table I may be rewritten with the combinators $E$ and $D_k$ listed below. These combinators are used to organize the definition so that each phrase is of the form $D_k(\alpha, \beta)$ or $E(\alpha, \beta)$, where $\alpha$ and $\beta$ have no free variables. The $E$ combinator is used to extend an environment. $D_k$ steers an environment to the first sub-term ($\alpha$) and steers an environment and a continuation to the second sub-term ($\beta$):

$$D_k(\alpha, \beta) = \lambda \rho e x_1 \cdots x_k \cdot \alpha \rho (\beta \rho e x_1 \cdots x_k)$$

$$E(\alpha, \beta) = \lambda \rho e \cdot \alpha \rho (\lambda \rho \cdot \beta \rho e).$$

Consider the let expression. The goal is to rewrite it into an expression of the form $E(\delta'[e_1], f)$ where $f$ evaluates the body ($e_2$) in an extended environment. Expanding $E(\delta'[e_1], f)$ yields $\lambda \rho e \cdot \delta'[e_1] \rho (\lambda \rho \cdot f \rho e)$. The functionality of $\delta$ mandates that its continuation be a function of an expressed value. Therefore, we define an auxiliary function $\text{bind}$ to accept the expressed value generated by $\delta'[e_1]$ and we use $D_0$ to steer the expressed value to the auxiliary. The combinator clause below is equivalent to the clause for the let expression in Table I:

$$\delta'[\text{let } i = e_1 \text{ in } e_2] = E(D_0(\delta'[e_1], \text{bind } i), \delta'[e_2])$$

$$\text{bind } i = \lambda \rho \chi v \cdot \chi (\text{ext-rho } i \rho v).$$

The compilers developed in (Wand, 1980, 1982a, 1982b, 1983) produce rotated trees which represent the combinator clauses. Such trees have
internal nodes labeled with $D_k$s and $E$s. A tree whose root is $D_k$ with a left child $\alpha$ and a right child $\beta$ is denoted by $[D_k, \alpha, \beta]$. For example, the tree for the expression let $x = \text{int } 1$ in $x$ is $[E, [D_0, [\text{int-const } 1], \text{bind } x], \text{lookup } x]$. Each leaf represents the auxiliary function with the corresponding name. For instance, the machine instruction $\text{lookup } x$ represents the expression $\text{lookup } x$.

In (Wand, 1980, 1982a, 1982b, 1983), the trees are rotated to obtain a sequence of machine instructions. A rotated tree can be executed in a simple iterative fashion by an abstract machine derived from the original definition. In this paper, we focus on compiler correctness and will not show the derivation of a machine for EL. Tree rotation is based on associative and distributive properties of some of the combinators; compiler correctness depends on showing that these properties preserve the semantics of the language. For more details on the role of rotation and on machine construction, the reader is invited to read (Wand, 1980, 1982a, 1982b, 1983).

The clause for an addition expression can be rewritten using the technique demonstrated above for the let expression. It is more complex because the clause contains several run-time type-checking operations and binary operations. The auxiliary functions $\text{int-add}$, $\text{real-add}$, $\text{bool-or}$, and $\text{coerce}$,
defined in the Appendix, are necessary but not sufficient for rewriting the addition clause into combinator form. New auxiliaries called \texttt{int?}, \texttt{real?}, and \texttt{bool?} must be designed. The purpose of \texttt{int?}, \texttt{real?}, or \texttt{bool?} is to test an expressed value for membership in the subdomain Int, Real, or Bool, respectively. The result of the test will determine which primitive operation (\texttt{int-add}, \texttt{real-add}, or \texttt{bool-or}) to perform and also whether or not a coercion is necessary.

If we were to rewrite the addition clause in Table I using \(D_\lambda\), \texttt{int?}, \texttt{real?}, \texttt{bool?}, and so forth, then the compiler derived from this clause would produce a tree of the form shown in Fig. 1. The tree contains three conditional subtrees: one labeled \texttt{int?}, one labeled \texttt{real?}, and one labeled \texttt{bool?}. These nodes represent the run-time operations which check the type of the value of \(x\). Each of these subtrees contains three conditional subtrees to test the type of \(y\). The choice of which subtree to execute is based on the results of the tests. Only a portion of the generated code would be executed.

It is possible to apply the methodology of (Wand, 1980, 1982a, 1982b, 1983) to the factored clause for addition appearing in Table III. In fact, only the dynamic portion of the factored clause need be rewritten using combinators. By expressing type analysis as a static expression, we have reduced the dynamic semantics of the language and consequently, the target code is more compact. For example, if static analysis reveals that \(x\) is real and \(y\) is integer, then the compiler would construct the tree shown in Fig. 2 for the expression \((x + y)\). No type-checking instructions are generated and the code contains all necessary coercion instructions. Actually, the compiler described below produces a linear form of the tree in Fig. 2, as explained in Section 4.

![Fig. 2. Optimized code for addition expression \((x + y)\).]
4. Compiling Expressions

The compiling algorithm for EL is presented in Table IV. The function \( \varepsilon_{\text{compile}} \) takes an expression and a type environment and returns code which correctly represents the meaning of the expression. Type checking is done

| Table IV |
| EL Compiler |

\[
\varepsilon_{\text{compile}} : \text{Exp} \rightarrow \text{Type-Env} \rightarrow \text{Code}_{\text{Exp}}
\]

\[
\varepsilon_{\text{compile}}[\text{int } c] = [\text{int-const } c]
\]

\[
\varepsilon_{\text{compile}}[\text{real } e] = [\text{real-const } e]
\]

\[
\varepsilon_{\text{compile}}[\text{bool } e] = [\text{bool-const } e]
\]

\[
\varepsilon_{\text{compile}}[e_1 + e_2] =
\]

let \( \text{code}_1 = \varepsilon_{\text{compile}}[e_1] \) in

\[
\begin{align*}
\text{case } \varepsilon_1[\text{e}_1] &\varepsilon_2[\text{e}_2] \\
\text{of}
\end{align*}
\]

\[
\begin{align*}
\text{"untyped" : } [\text{type-error error},] \quad \text{else : } [\text{lookup } 1]
\end{align*}
\]

\[
\begin{align*}
\varepsilon_{\text{compile}}[\text{let } i = e_1 \text{ in } e_2] =
\]

let \( \text{code}_1 = \varepsilon_{\text{compile}}[e_1] \) in

\[
\begin{align*}
\text{case } \varepsilon_1[\text{e}_1] &\varepsilon_2[\text{e}_2] \\
\text{of}
\end{align*}
\]

\[
\begin{align*}
\text{"untyped" : } [\text{type-error error},] \quad \text{else : } [\text{case } \varepsilon_1[\text{e}_1] &\varepsilon_2[\text{e}_2] \\
\text{of}
\end{align*}
\]

\[
\begin{align*}
\text{"int" : } \left[\text{let code}_2 = \varepsilon_{\text{compile}}[e_2] \text{ in } \right. \\
\text{\quad } \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \text{\quad \tex}
before any code is generated. Hence, the compiler produces linear trees of instructions and no type-checking instructions appear in the compiler output.

The compiler produces trees which directly correspond to the combinator portion of the factored clauses. Compiler correctness is straightforward to prove (Montenyohl, 1986). Theorem 4.3 says that the function which corresponds to the target code is equivalent to the original definition of EL. Additional notation is used to state the theorem: let \( \phi \) be the function which maps target code to the corresponding combinator function. For example,

\[
\phi([D_0 \text{ [lookup } x \text{ ] } [D_1 \text{ [lookup } y \text{ ] int-add ]}])
= D_0(\text{lookup } x, D_1(\text{lookup } y, \text{int-add})).
\]

The proof depends on three facts. First, the rotation function which is used to linearize the trees is semantic-preserving. Propositions 4.1 and 4.2, stated below, describe how the combinator terms may be rotated without disturbing the meaning of an expression. Second, the combinators preserve the semantics of the language. This can be proved by expanding combinator terms into the factored clauses. Third, Theorem 2.3 states that the factored clauses are equivalent to the original definition of EL.

The methodology summarized above is applicable to other language constructs. In the next section, the compiler is enhanced to translate assignment statements and loops. The same steps are taken to convert a continuation semantics for assignment and loops into a compiler specification.

**Proposition 4.1.**

\[
D_k(D_p(\alpha, \beta), \gamma) = D_{k+p}(\alpha, D_k(\beta, \gamma)).
\]

**Proof.**

\[
\begin{align*}
D_k(D_p(\alpha, \beta), \gamma) &= \lambda \rho \kappa x_1 \cdots x_k \cdot D_p(\alpha, \beta)\rho(\gamma \rho \kappa x_1 \cdots x_k) \\
&= \lambda \rho \kappa x_1 \cdots x_k \cdot \lambda x_{k+1} \cdots x_{k+p} \\
&\quad \cdot \alpha \rho(\beta \rho(\gamma \rho \kappa x_1 \cdots x_k)x_{k+1} \cdots x_{k+p}) \\
&= \lambda \rho \kappa x_1 \cdots x_{k+p} \cdot \alpha \rho(D_k(\beta, \gamma) \rho \kappa x_1 \cdots x_{k+p}) \\
&= D_{k+p}(\alpha, D_k(\beta, \gamma)).
\end{align*}
\]

**Proposition 4.2.**

\[
E(D_0(\alpha, \beta), \gamma) = D_0(\alpha, E(\beta, \gamma)).
\]
Proof.

\[ E(D_0(a, \beta), \gamma) \]

\[ = \lambda \rho \chi \cdot D_0(a, \beta) \rho(\lambda \rho' \cdot \gamma \rho' \chi) \]

\[ = \lambda \rho \chi \cdot \alpha \rho(\beta \rho(\lambda \rho' \cdot \gamma \rho' \chi)) \]

\[ = \lambda \rho \chi \cdot \alpha \rho(E(\beta, \gamma)) \]

\[ = D_0(a, E(\beta, \gamma)). \]

Theorem 4.3. Let \( \delta \) be the function defined in Table I. Let \( \delta_i \) be the function defined in Table II. Let \( \delta_{cmpb} \) be the function defined in Table IV. Let \( \langle \tau, \rho \rangle \) be type consistent. For all \( e \in \text{Econt} \) and for all \( e \in \text{Exp} \),

if \( \delta_i[e] \tau \neq \text{"untyped"} \) then \( \phi(\delta_{cmpb}[e] \tau) \rho \varepsilon = \delta[e] \rho \varepsilon. \)

5. An Imperative Language with Loops

Table V contains the syntax and semantics for two kinds of statements: assignment and iteration. Adding assignment to EL requires modifications to the semantic domains and equations. Identifiers will now denote locations (instead of values) and the store maps location to values. The semantics of an identifier expression is different; the function \( \delta \) calculates the stored value associated with an identifier using the auxiliary functions int-fetch, real-fetch, and bool-fetch. Also, the treatment of let expressions is different. The environment and store must be updated with the new binding before the body can be evaluated. In the tables which follow, new dynamic, static, and factored clauses for identifier and let expressions will be given. For all other kinds of expressions, the semantics is the same as the equations in Table I through Table IV and so the clauses will not be displayed again.

Since EL has three types of values, there are three kinds of locations (Int-Loc, Real-Loc, and Bool-Loc) and the store is a tuple of three functions (Int-Store, Real-Store, and Bool-Store). For each type of location, one store function maps the location to its appropriate stored value (Int-Svalue, Real-Svalue, and Bool-Svalue).

The clause for an assignment statement tests the type of the identifier and the type of the expression to determine if a coercion is needed and to select which update function to apply. The auxiliary function int-update is defined in the Appendix; the definition of real-update and bool-update are similar and therefore not included. The meaning of a loop is defined as the least fixed point of a function; the body of the loop is executed only if the
MONTENYHOHL AND WAND

TABLE V

Statements in EL

Syntax

\[
\text{(Stmt)} ::= (\text{Ide}) ::= (\text{Exp}) \mid \text{while} (\text{Exp}) \text{ do} (\text{Stmt})
\]

Semantics

\[
\begin{align*}
\text{Location} &= \text{Int-Loc} + \text{Real-Loc} + \text{Bool-Loc} \\
\text{Dvalue} &= \text{Location} + \{ \text{‘undeclared’} \} \\
\text{Env} &= \text{Id} \rightarrow \text{Dvalue} \\
\text{Int-Svalue} &= \text{Int} + \{ \text{‘uninitialized int-lot’} \} \\
\text{Real-Svalue} &= \text{Real} + \{ \text{‘uninitialized real-lot’} \} \\
\text{Bool-Svalue} &= \text{Bool} + \{ \text{‘uninitialized bool-lot’} \} \\
\text{Int-Store} &= \text{Int-Loc} \rightarrow \text{Int-Svalue} \\
\text{Int-Store} &= \text{Real-Loc} \rightarrow \text{Real-Svalue} \\
\text{Bool-Store} &= \text{Bool-Loc} \rightarrow \text{Bool-Svalue} \\
\text{Ccont} &= \text{Store} \rightarrow \text{Answer} \\
\text{Econt} &= \text{Evalue} \rightarrow \text{Ccont} \\
\text{Lcont} &= \text{Location} \rightarrow \text{Ccont} \\
\text{Dcont} &= \text{Env} \rightarrow \text{Store} \rightarrow \text{Ccont} \\
\text{S} : \text{Stmt} \rightarrow \text{Env} \rightarrow \text{Ccont} \rightarrow \text{Ccont} \\
\text{E} : \text{Exp} \rightarrow \text{Env} \rightarrow \text{Econt} \rightarrow \text{Ccont} \\
\text{L} : \text{Ide} \rightarrow \text{Env} \rightarrow \text{Lcont} \rightarrow \text{Ccont}
\end{align*}
\]

boolean expression evaluates to true. The syntax and semantics of expressions should be expanded to include booleans. However, this information is not given in Table V because it adds nothing new to the discussion.

The clauses in Table V contain dynamic type-checking operations which
CORRECT STATIC ANALYSIS

165

TABLE VI
Static Semantics

Msg = {'ok', 'error'}

\[ S_t : Stmt \rightarrow Type-Env \rightarrow Msg \]
\[ E_t : Exp \rightarrow Type-Env \rightarrow Type \]
\[ L_t : Ide \rightarrow Type-Env \rightarrow Type \]

\[ S_t[i := e_t]r = \]
\[ L_t[i]r = \begin{cases} 'int' & \Rightarrow E_t[i]r = 'int' \Rightarrow 'ok', 'error', \\
                           'real' & \Rightarrow E_t[i]r = 'real' \Rightarrow 'ok', 'error', \\
                           'bool' & \Rightarrow E_t[i]r = 'bool' \Rightarrow 'ok', 'error', \\
                           'error' & \end{cases} \]

\[ S_t[\text{while } e \text{ do } s]r = \]
\[ E_t[s]r = 'bool' \Rightarrow S_t[i]r, 'error' \]

\[ L_t[i]r = L_t[i]r \]
\[ L_t[\text{let } i = e_1 \text{ in } e_2]r = \]
\[ E_t[e_1]r = 'int' \Rightarrow E_t[e_2]((\text{ext-tau } i \tau \text{'int'}) \]
\[ E_t[e_1]r = 'real' \Rightarrow E_t[e_2]((\text{ext-tau } i \tau \text{'real'}) \]
\[ E_t[e_1]r = 'bool' \Rightarrow E_t[e_2]((\text{ext-tau } i \tau \text{'bool'})\text{'untyped'} \]
\[ L_t[i]r = \tau i \]

will be replaced by static tests. The static semantics for statements is given in Table VI. The function \( S_t \) maps a statement to a message; if the statement is free of type errors, its static value is “ok,” otherwise its static value is “error.” A loop is well typed provided the body has no type errors and the type of the expression is boolean. The static analysis described in Table VI may replace the run-time analysis provided the consistency property stated below holds between the static environment, the dynamic environment, and the store.

The first two items in this property are retained from the previous type consistency property. The third item states the necessary relationship between the static environment, dynamic environment and the store: if an identifier is statically bound to type “int,” “real,” or “bool” then the dynamic environment binds that identifier to an element of \( \text{Int-Loc} \), \( \text{Real-Loc} \), or \( \text{Bool-Loc} \), respectively, and the value stored in the location is a member of \( \text{Int} \), \( \text{Real} \), or \( \text{Bool} \), respectively.

Formal statements regarding type consistency are found in Lemmas 5.1 and 5.2 stated below. Lemma 5.1 says that consistency is preserved by the auxiliary functions which operate on the environments and the store. The proof of Lemma 5.1 is straightforward, by definition. Lemma 5.2 asserts that consistency is preserved by the evaluation of a well-typed identifier, expression, or statement. Therefore, type consistency is presented throughout the execution of a well-typed program. The notation
introduced in Lemma 1.1 is extended for the statement of Lemma 5.2: the expression $\text{consis?'~}(\tau, \rho, \sigma)$ is true if $\langle \tau, \rho, \sigma \rangle$ are type consistent. Lemma 5.2 is proved using Lemma 5.1 in a manner similar to the proof given for Lemma 1.1.

**Type Consistency Property.** Let $\tau \in \text{Type-Env}$, $\rho \in \text{Env}$, and $\sigma \in \text{Store}$. $\langle \tau, \rho, \sigma \rangle$ are type consistent if and only if for all $i \in \text{Ide}$,

(i) $\rho_i \neq \perp_{\text{D-value}}$ and $(\rho_i)|_{\text{Location}} \neq \perp_{\text{Location}}$

(ii) $\tau_i = \text{"untyped"} \iff \rho_i = \text{"undeclared"}$

(iii) the following propositions are equivalent

- $\tau_i = \text{"int"}$
- $\text{isInt-Loc? } [(\rho_i)|_{\text{Location}}]$
- $\text{isInt? } (\sigma \downarrow_1 [(\rho_i)|_{\text{Location}}|\text{Int-Loc}])$

(iv) the following propositions are equivalent

- $\tau_i = \text{"real"}$
- $\text{isReal-Loc? } [(\rho_i)|_{\text{Location}}]$
- $\text{isReal? } (\sigma \downarrow_2 [(\rho_i)|_{\text{Location}}|\text{Real-Loc}])$

(v) the following propositions are equivalent

- $\tau_i = \text{"bool"}$
- $\text{isBool-Loc? } [(\rho_i)|_{\text{Location}}]$
- $\text{isBool? } (\sigma \downarrow_3 [(\rho_i)|_{\text{Location}}|\text{Bool-Loc}])$

**LEMMA 5.1.** Let $\langle \tau, \rho, \sigma \rangle$ be type consistent. For all $i \in \text{Ide}$,

(i) if $v \in \text{Int}$, and $l \in \text{Int-Loc}$, then

$\langle \text{ext-tau } i \text{ it } \text{"int"}, \text{ext-rho } ip[l \in \text{Location}] \rangle$ and

$\langle [\lambda \cdot \cdot \cdot (y = l) \Rightarrow [v \text{ inInt-Svalue}], (\sigma \downarrow_1) y], (\sigma \downarrow_2, \sigma \downarrow_3) \rangle$

are type consistent.

(ii) if $v \in \text{Real}$ and $l \in \text{Real-Loc}$, then

$\langle \text{ext-tau } i \text{ it } \text{"real"}, \text{ext-rho } ip[l \in \text{Location}] \rangle$ and

$\langle \sigma \downarrow_1, [\lambda \cdot \cdot \cdot (y = l) \Rightarrow [v \text{ inReal-Svalue}], (\sigma \downarrow_2) y], \sigma \downarrow_3 \rangle$

are type consistent.

(iii) if $v \in \text{Bool}$ and $l \in \text{Bool-Loc}$, then

$\langle \text{ext-tau } i \text{ it } \text{"bool"}, \text{ext-rho } ip[l \in \text{Location}] \rangle$ and

$\langle \sigma \downarrow_1, \sigma \downarrow_2, [\lambda \cdot \cdot \cdot (y = l) \Rightarrow [v \text{ inBool-Svalue}], (\sigma \downarrow_3) y] \rangle$

are type consistent.

**LEMMA 5.2.** Let $\langle \tau, \rho, \sigma \rangle$ be type consistent. Let error $\in \text{Answer}$. For all $\tau \in \text{Type-Env}$, $\rho \in \text{Dynamic-Env}$ and $\sigma \in \text{Store}$, let $\text{consis?'~} \langle \tau, \rho, \sigma \rangle \iff \langle \tau, \rho, \sigma \rangle$ are type consistent.
(i) For all \( i \in \text{Ide} \) and \( \eta \in \text{Lcont} \), if \( \mathcal{L}_i[i] \tau \neq \text{"untyped"} \) then
\[
\mathcal{L}_i[i] \rho(\lambda l \cdot \text{consis? } \langle \tau, \rho, \sigma' \rangle \Rightarrow \eta l \sigma', \text{error}) \sigma = \mathcal{L}_i[i] \rho \eta \sigma.
\]

(ii) For all \( e \in \text{Exp} \) and \( \varepsilon \in \text{Econt} \), if \( \mathcal{E}_e[e] \tau \neq \text{"untyped"} \) then
\[
\mathcal{E}_e[e] \rho(\lambda v \cdot \text{consis? } \langle \tau, \rho, \sigma' \rangle \Rightarrow \varepsilon v \sigma', \text{error}) \sigma = \mathcal{E}_e[e] \rho \varepsilon \sigma.
\]

(iii) For all \( s \in \text{Stmt} \) and \( \kappa \in \text{Ccont} \), if \( \mathcal{S}_s[s] \tau = \text{"ok"} \) then
\[
\mathcal{S}_s[s] \rho(\lambda \sigma' \cdot \text{consis? } \langle \tau, \rho, \sigma' \rangle \Rightarrow \kappa \sigma', \text{error}) \sigma = \mathcal{S}_s[s] \rho \kappa \sigma.
\]

Theorem 5.3 allows for the correct and equivalent replacement of runtime tests with static tests in expressions. Part (i) describes the correct substitution for tests involving locations. Testing a location for membership in Int-Loc, Real-Loc, or Bool-Loc is equivalent to asking if the corresponding identifier is statically bound to "int," "real," or "bool," respectively. Part (ii) states the correctness of replacing run-time tests of an expressed value with the static tests of the expression. The proof of part (i) comes directly from the Type Consistency Property. The proof of part (ii) is done by structural induction on expressions and uses Lemma 5.2 (Montenyohl, 1986).

**Theorem 5.3.** Let \( \langle \tau, \rho, \sigma \rangle \) be type consistent.

(i) For all \( i \in \text{Ide} \) and for all \( f_j \in \text{Lcont}, 1 \leq j \leq 4 \),

if \( \mathcal{L}_i[i] \tau \neq \text{"untyped"} \) then
\[
\mathcal{L}_i[i] \rho(\lambda l \cdot \text{isInt-Loc? } l : f_1 l
\hspace{1cm}
isReal-Loc? l : f_2 l
\hspace{1cm}
isBool-Loc? l : f_3 l) \sigma
\]
\[
= \mathcal{L}_i[i] \rho(\lambda l \cdot \mathcal{L}_i[i] \tau = \text{"int"} \rightarrow f_1 l
\hspace{1cm}
\mathcal{L}_i[i] \tau = \text{"real"} \rightarrow f_2 l
\hspace{1cm}
\mathcal{L}_i[i] \tau = \text{"bool"} \rightarrow f_3 l)
\sigma.
\]

(ii) For all \( e \in \text{Exp} \) and for all \( f_j \in \text{Econt}, 1 \leq j \leq 4 \),

if \( \mathcal{E}_e[e] \tau \neq \text{"untyped"} \) then
\[
\mathcal{E}_e[e] \rho(\lambda v \cdot \text{isInt? } v : f_1 v
\hspace{1cm}
isReal? v : f_2 v
\hspace{1cm}
isBool? v : f_3 v) \sigma
\]
\[
= \mathcal{E}_e[e] \rho(\lambda v \cdot \mathcal{E}_e[e] \tau = \text{"int"} \rightarrow f_1 v
\hspace{1cm}
\mathcal{E}_e[e] \tau = \text{"real"} \rightarrow f_2 v
\hspace{1cm}
\mathcal{E}_e[e] \tau = \text{"bool"} \rightarrow f_3 v)
\sigma.
\]
Factored clauses specifying the meaning of an assignment statement and a loop appear in Table VII. Theorem 5.4 states that the factored clauses are equivalent to the original dynamic clauses. The proof for expressions and for the assignment statement is a direct result of the definitions of $\delta$, $\mathcal{I}$, $\delta'$, $\mathcal{I}'$, Lemma 5.2, the induction hypothesis, and the factorization corollary.
\[
S_{\text{stmt}} : \text{Stmt} \rightarrow \text{Type-Env} \rightarrow \text{Code}_{\text{stmt}}
\]
\[
E_{\text{stmt}} : \text{Exp} \rightarrow \text{Type-Env} \rightarrow \text{Code}_{\text{exp}}
\]
\[
L_{\text{stmt}} : \text{Id} \rightarrow \text{Type-Env} \rightarrow \text{Code}_{\text{id}}
\]
\[
S_{\text{stmt}}[l := e] =
\]
\[
\text{let } \text{code}_l = E_{\text{stmt}}[e] \text{r and } \text{code}_e = S_{\text{stmt}}[e] \text{r}
\]
\[
\text{case } L_{\text{stmt}}[l] \text{r of}
\]
\[
\text{'int':}
\]
\[
\text{case } L_{\text{stmt}}[e] \text{r of}
\]
\[
\text{'int':}
\]
\[
\text{case } L_{\text{stmt}}[e] \text{r of}
\]
\[
\text{'real':}
\]
\[
\text{case } L_{\text{stmt}}[e] \text{r of}
\]
\[
\text{'bool':}
\]
\[
\text{case } L_{\text{stmt}}[e] \text{r of}
\]
\[
\text{'untyped':}
\]
\[
\text{case } L_{\text{stmt}}[e] \text{r of}
\]
\[
\text{'error':}
\]
\[
E_{\text{stmt}}[i] =
\]
\[
\text{let } \text{code}_e = E_{\text{stmt}}[e] \text{r}
\]
\[
\text{case } E_{\text{stmt}}[i] \text{r of}
\]
\[
\text{'int':}
\]
\[
\text{case } E_{\text{stmt}}[i] \text{r of}
\]
\[
\text{'real':}
\]
\[
\text{case } E_{\text{stmt}}[i] \text{r of}
\]
\[
\text{'bool':}
\]
\[
\text{case } E_{\text{stmt}}[i] \text{r of}
\]
\[
\text{'untyped':}
\]
\[
\text{case } E_{\text{stmt}}[i] \text{r of}
\]
\[
\text{L_{stmt}}[\text{let } i = e; \text{ in } e'] =
\]
\[
\text{let } \text{code}_i = E_{\text{stmt}}[i] \text{r}
\]
\[
\text{case } E_{\text{stmt}}[e] \text{r of}
\]
\[
\text{L_{stmt}}[\text{if } t \text{ then } e_1 \text{ else } e_2] =
\]
\[
\text{case } L_{\text{stmt}}[t] \text{r of}
\]
\[
\text{Rotation Function}
\]
\[
\text{not } [b] = ([b] \sigma) \theta \text{ not } [b] \sigma \theta \text{ [test } \beta \text{ return}] \text{ }) \}
\]
\[
\text{not } [b] = [b] \text{ not } [b] \text{ if } \alpha \neq [b] \text{ and } \alpha \neq [b] \text{ return } [b] \text{ else } [b] \text{ return } [b]
\]
\[
\text{not } [t] = t \text{ if } t \text{ is a leaf}
\]
Each step follows the logic of the proof of Theorem 2.3. The proof for loops uses a fixed-point argument as shown in the Appendix.

**Theorem 5.4.** Let \( \langle \tau, \rho, \sigma \rangle \) be type consistent. Let \( \mathcal{S}, \mathcal{L}, \) and so forth be the functions defined in Table V. Let \( \mathcal{S'}, \mathcal{L'}, \) and so forth be the functions defined in Table VI. Let \( \mathcal{S''}, \mathcal{L''}, \) and so forth be the functions defined in Table VII.

(i) For all \( i \in \text{Ide} \) and \( \eta \in \text{Lcont}, \)

\[
\text{if } \mathcal{L}([i] \tau \neq \text{"untyped"} \text{ then } \mathcal{L}([i] \rho \eta \sigma = \mathcal{L}'([i] \tau \rho \eta \sigma) .}
\]

(ii) For all \( e \in \text{Exp} \) and \( e \in \text{Econt}, \)

\[
\text{if } \mathcal{E}([e] \tau \neq \text{"untyped"} \text{ then } \mathcal{E}([e] \rho \sigma = \mathcal{E}'([e] \tau \rho \sigma) .}
\]

(iii) For all \( s \in \text{Stmt} \) and \( \kappa \in \text{Ccont}, \)

\[
\text{if } \mathcal{S}([s] \tau = \text{"ok"} \text{ then } \mathcal{S}([s] \rho \kappa \sigma = \mathcal{S}'([s] \tau \rho \kappa \sigma) .}
\]

Table VIII contains the compiling algorithm for a statement, an identifier, and a let expression. Addition expressions are compiled as stated in Table IV. Theorem 5.7 states that each compiling function generates code which correctly represents the dynamic denotation of each identifier, expression, and statement. Just as with Theorem 4.3, compiler correctness depends on showing that the combinators and the rotations are semantic preserving. For expressions and the assignment statement, expanding the combinators yields the factored clauses. The rotation equations relevant to expressions and assignment have already been proved correct in Section 4. Showing that compilation is correct for loops is the only non-trivial part of the proof (Montenyohl, 1986).

The loop clause is rewritten using \( D_{\kappa} \) and label where label is a syntactic abbreviation defined as \( \text{label}(\theta, \beta) = \text{fix}(\lambda \theta \cdot \beta) \). The factored clause for a well-typed loop is in the form \( \lambda \rho \kappa \cdot \text{fix}(\lambda \theta \cdot \mathcal{M} \rho \kappa \theta) \) and the combinator clause for a loop has the form \( \text{fix}(\lambda \theta' \cdot \lambda \rho \kappa \cdot \mathcal{M}' \rho \kappa \theta') \). Proposition 5.6, stated below, shows how the first expression may be transformed to the second expression. This proposition is a variant of one presented in (Wand, 1983). A fixed-point proof of Proposition 5.5 is given in (Montenyohl, 1986). The code for loops is rotated according to Proposition 5.6. The proof is an immediate consequence of the definitions.

**Proposition 5.5.** Let \( e \in \text{Exp}, \) and \( s \in \text{Stmt}; \)

\[
\text{fix}(\lambda \theta' \cdot D_{\kappa}(\mathcal{E}'([e] \tau, \text{test}(D_{\kappa}(\mathcal{S}'([s] \tau, \theta'), \text{return})))) = \lambda \rho \kappa \cdot \text{fix}(\lambda \theta \cdot \mathcal{E}'([e] \tau \rho \sigma, \text{test}(\lambda \rho \kappa \cdot \mathcal{S}'([s] \tau \rho \theta), \text{return}) \rho \kappa)) .
\]
Proposition 5.6. Let \( \tau, \rho, \sigma \) be type consistent. If \( \mathcal{S} \left[ \begin{array}{c} \text{while } e \text{ do } \end{array} \right] s \] \( \tau = \text{"ok"} \), then for all \( \kappa \in \text{Ccont} \),

\[
D_0(\mathcal{S} \left[ \begin{array}{c} \text{while } e \text{ do } \end{array} \right] s \] \( \tau, \gamma \) \rho \kappa \sigma
= \text{label}(\theta, D_0(\mathcal{S} \left[ \begin{array}{c} e \end{array} \right] \tau, \text{test}(D_0(\mathcal{S} \left[ \begin{array}{c} s \end{array} \right] \tau, \theta), \gamma))) \rho \kappa \sigma.
\]

Theorem 5.7. Let \( \tau, \rho, \sigma \) be type consistent.

(i) For all \( i \in \text{Ide} \) and \( \eta \in \text{Lcont} \),

\[
\text{if } \mathcal{L}_i[i] \tau \neq \text{"untyped" } \text{then } \phi(\mathcal{L}_{\text{emplr}}[i] \tau) \rho \eta \sigma = \mathcal{L}[i] \rho \eta \sigma.
\]

(ii) For all \( e \in \text{Exp} \) and \( \epsilon \in \text{Econt} \),

\[
\text{if } \mathcal{S}_i[e] \tau \neq \text{"untyped" } \text{then } \phi(\mathcal{S}_{\text{emplr}}[e] \tau) \rho \epsilon \sigma = \mathcal{S}[e] \rho \epsilon \sigma.
\]

(iii) For all \( s \in \text{Stmt} \) and \( \kappa \in \text{Ccont} \),

\[
\text{if } \mathcal{S}_i[s] \tau = \text{"ok" } \text{then } \phi(\mathcal{S}_{\text{emplr}}[s] \tau) \rho \kappa \sigma = \mathcal{S}[s] \rho \kappa \sigma.
\]

6. Related Work

The rearrangement phase of our methodology is similar to a technique called staging transformations presented in (Jorring and Scherlis, 1986). They derive fragments of compiler from interpreters by shifting computations to an earlier stage where they may be carried out less frequently.

Our approach to optimization is different from that used by Hudak and Kranz. In (Hudak and Kranz, 1983), an efficient implementation for a lazy functional language is presented. Code optimization is performed by reducing the combinator terms; the optimizations are described as tree-rewriting rules. Our compiler produces optimized trees using static analysis and once the trees are constructed, they are not pruned.

The methodology presented here does not use abstract evaluators for static analysis as presented in (Donzeau-Gouge, 1980; Pleban and Muchnick, 1980; Nielson, 1982, 1985; Barbuti and Martelli, 1984). An abstract evaluator describes static properties in terms of a non-standard definition in which information is associated with program points (Occurrences) in order to deduce things about the run-time behavior of the program. The static semantics specified in this paper is defined over a small set of domains, none of which is the domain of Occurrences. Furthermore, the static analysis does not involve any run-time objects. As a result, the static semantics given here is very straightforward; only a few domains are involved and these domains correspond directly to data structures used by
a compiler. The static environment represents the compile-time symbol table which maps identifiers to static values.

Other static information could be included in the static environment in order to solve other flow analysis problems. In (Montenyohl, 1986), constant-folding is addressed. The static environment is enhanced to keep track of constant-valued identifiers and expressions; compile-time computations are performed whenever the value of the operands are constants.

The static semantics for EL is specified separately from the dynamic meaning of the language as a function of syntax and a static environment and then static type-checking is incorporated into the semantics of EL to replace run-time checks. This is different from the technique presented in (Barbuti and Martelli, 1984), where a static semantics is extracted from a standard definition of the source language. Regardless of the approach, we share a common goal which is to establish a strong connection between static analysis and dynamic behavior.

In our approach, the key to incorporating the static definition into the dynamic definition is the Type Consistency Property. This property reflects the necessary dependence between static and dynamic environments and the store. A similar sort of assertion appears in (Milner, 1978).

Increasing the complexity of the source language requires additional work in order to derive the compiler and to prove it correct. In (Montenyohl, 1986), a compiler for a block-structured, statement-oriented language is designed. The transformation of the dynamic definition to the compiler specification follows the same step explained here. However, additional lemmata are necessary to show that type-consistency is preserved by other program constructs. In particular, the language allows sequences of declarations and statements, any of which may alter the environment and/or store. Therefore, it must be shown that type consistency is preserved by all defined operations on the environment and store.

7. CONCLUSION

We have shown how to extend the combinator-based methodology in (Wand, 1980, 1982a, 1982b, 1983) to incorporate static analysis and code optimization into a compiler specification. A continuation semantics, containing dynamic type checks, has been transformed into a compiler specification containing static-type analysis. Transformations include replacing run-time computations with compile-time computations and rearranging clauses so that the dynamic meaning of a construct depends on static analysis. The key to the correctness proof is to show that each transformation preserves the original definition.
Auxiliary Functions

terminate, int-const, real-const, bool-const

[definition deferred until implementation-time]

ext-rho: Ide → Env → Eval → Env

ext-rho = λiplx.(x = i) ⇒ [v in Dvalue], ρx

ext-tau: Ide → Type-Env → Type → Type-Env

ext-tau = lixtx.(x = i) ⇒ t, τx

lookup i = λpe.ρi = “undeclared” ⇒ terminate error1, ε[ρi in Eval]

int-add = λρev1ν2.ε([ν1 in Int] + [ν2 in Int]) in Eval

real-add = λρev1ν2.ε([ν1 in Real] + [ν2 in Real]) in Eval

bool-or = λρev1ν2.ε([ν1 in Bool] + [ν2 in Bool]) in Eval

coerce = λρev.ε[v in Int in Real in Eval]

int-update: Env → Ccont → Location → Eval → Ccont

int-update = λρkλvσ.

let loc = l|Int-Loc and val = v|Int in Int-Svalue

in κ<[λy.(y = loc) ⇒ val, (σ ↓ l)y], σ ↓ 2, σ ↓ 3>

test(x, β): Env → Ccont → Eval → Ccont

test(x, β) = λρkv.[v in Bool] ⇒ xρκ, βρκ

return: Env → Ccont → Ccont

return = λρkκ

int-fetch: Env → Econt → Location → Store → Answer

int-fetch = λρvσ.

let v = (σ ↓ l)[l|Int-Loc] in v = “uninitialized int-lot” ⇒ terminate error3 σ, ε[v in Int in Eval] σ

int-bind: Ide → Env → Dcont → Eval → Ccont

int-bind = λρvσ.

let loc = new-intloc σ

let ρ’ = ext-rho ip[loc in Location] and

let σ’ = <[λx.x = loc ⇒ v|Int in Int-Svalue, (σ ↓ l)x], σ ↓ 2, σ ↓ 3>

in ρ’σ’

lookup: Ide → Env → Lcont → Ccont

lookup = λipη.η[(ρi)|Location]

ew-intloc: Store → Int-Loc yields an unused location from Int-Store

Error Messages

error1 = “unbounded identifier”

error2 = “incompatible types in addition expression”

error3 = “untyped expression in binding”

error4 = “untyped expression in let body”

error5 = “untyped operand in addition expression”
error_6 = “incompatible expression in assignment”
error_7 = “non-boolean expression in loop”
error_8 = “uninitialized int-loc”
error_9 = “untyped expression in assignment”
error_10 = “untyped identifier in assignment”
error_11 = “untyped loop body”

Proofs

Proof of Lemma 1.1. Let $\langle \tau, \rho \rangle$ be type consistent. Let $error \in \text{Answer}$,
$\varepsilon \in \text{Econt}$. Let $x \in \text{Ide}$ be arbitrary but fixed.

(i) For constant expressions, the proof is straightforward from the definitions.

(ii) Let $i \in \text{Ide}$. Assume $\varepsilon[i] \tau \neq \text{"untyped."}$ By definition, $\tau i \neq \text{"untyped."}$ By definition of $\varepsilon$,

$$\begin{align*}
\varepsilon[i] \rho(\lambda v \cdot \text{consis?} \langle (\text{ext-tau } x \tau[i] \tau), (\text{ext-rho } x \rho v) \rangle \Rightarrow ev, error) \\
= (\rho i = \text{"undeclared"}) \Rightarrow \text{terminate error},
\end{align*}$$

$(\lambda v \cdot \text{consis?} \langle (\text{ext-tau } x \tau[i] \tau), (\text{ext-rho } x \rho v) \rangle \Rightarrow ev, error)[(\rho i)|\text{Eval}]$

By consistency, $\tau i \neq \text{"untyped"}$ implies $\rho i \neq \text{"undeclared"}$, thus,

$$\begin{align*}
\varepsilon[i] \rho(\lambda v \cdot \text{consis?} \langle (\text{ext-tau } x \tau[i] \tau), (\text{ext-rho } x \rho v) \rangle \Rightarrow ev, error) \\
= \lambda v \cdot \text{consis?} \langle (\text{ext-tau } x \tau[i] \tau), (\text{ext-rho } x \rho v) \rangle \\
\Rightarrow ev, error)[(\rho i)|\text{Eval}] \\
= \text{consis?} \langle (\text{ext-tau } x \tau[i] \tau), (\text{ext-rho } x \rho[(\rho i)|\text{Eval}]) \rangle \\
\Rightarrow \varepsilon[(\rho i)|\text{Eval}], error
\end{align*}$$

By definition, for any $y \in \text{Ide},$

$$\begin{align*}
(\text{ext-tau } x \tau[i] \tau) y = \begin{cases} 
\tau i, & \text{if } x = y \\
\tau y, & \text{otherwise}
\end{cases}
\end{align*}$$

and

$$\begin{align*}
(\text{ext-rho } x \rho[(\rho i)|\text{Eval}]) y = \begin{cases} 
\rho i, & \text{if } x = y \\
\rho y, & \text{otherwise}
\end{cases}
\end{align*}$$

$\langle \tau, \rho \rangle$ are type consistent, implies $(\text{ext-tau } x \tau[i])$ and $(\text{ext-rho } x \rho[(\rho i)|\text{Eval}])$ are type consistent. Therefore,

$$\begin{align*}
\varepsilon[i] \rho(\lambda v \cdot \text{consis?} \langle (\text{ext-tau } x \tau[i] \tau), (\text{ext-rho } x \rho v) \rangle \Rightarrow ev, error) \\
= \varepsilon[(\rho i)|\text{Eval}] \\
= \varepsilon[i] \rho v.
\end{align*}$$

(iii) Let $i \in \text{Ide}$. Let $e_1, e_2 \in \text{Exp}$. Let $e = \text{let } i = e_1 \text{ in } e_2$. Let $\tau' = (\ext-
tau \texttt{it} [\mathcal{E} [\tau_1 \tau_2]]. Assume \(\mathcal{E}_1 [\tau_1 \tau_2] \neq \text{"untyped".}\) By definition, \(\mathcal{E}_1 [\tau_1 \tau_2] \neq \text{"untyped" and } \mathcal{E}_2 [\tau_1 \tau_2] \neq \text{"untyped."} \)

\[
\mathcal{E} [\tau] \rho (\lambda v \cdot \text{consis}? (\langle \text{ext-tau } \tau [\mathcal{E} [\tau_1 \tau_2]], (\text{ext-rho } \tau_2 \tau_3) \rangle) \Rightarrow \epsilon_v, \text{error})
\]

\[
= \mathcal{E} [\tau_1] \rho (\lambda v_1 \cdot \mathcal{E}_2 \rho (\text{ext-rho } \tau_2 \tau_3), (\langle \tau_2 \tau_3 \rangle \Rightarrow \epsilon_v, \text{error}))
\]

\textbf{definition of } \mathcal{E}_1

\[
= \mathcal{E} [\epsilon_1] \rho (\lambda v' \cdot \text{consis}? (\langle \text{ext-tau } \tau [\mathcal{E}_1 [\tau_1 \tau_2]], (\text{ext-rho } \tau_2 \tau_3) \rangle) \Rightarrow \epsilon_v, \text{error})
\]

\textbf{induction hypothesis on } \epsilon_1

\[
= \mathcal{E} [\epsilon_1] \rho (\lambda v' \cdot \text{consis}? (\langle \text{ext-tau } \tau [\mathcal{E}_1 [\tau_1 \tau_2]], (\text{ext-rho } \tau_2 \tau_3) \rangle) \Rightarrow \epsilon_v, \text{error})
\]

\textbf{definition of } \mathcal{E}_1

\[
= \mathcal{E} [\epsilon_1] \rho (\lambda v' \cdot \text{consis}? (\langle \text{ext-tau } \tau [\mathcal{E}_1 [\tau_1 \tau_2]], (\text{ext-rho } \tau_2 \tau_3) \rangle) \Rightarrow \epsilon_v, \text{error})
\]

\textbf{induction hypothesis on } \epsilon_2

\[
= \mathcal{E} [\epsilon_1] \rho (\lambda v' \cdot \text{consis}? (\langle \text{ext-tau } \tau [\mathcal{E}_1 [\tau_1 \tau_2]], (\text{ext-rho } \tau_2 \tau_3) \rangle) \Rightarrow \epsilon_v, \text{error})
\]

\textbf{induction hypothesis on } \epsilon_2

\[
= \mathcal{E} [\epsilon_1] \rho (\lambda v' \cdot \text{consis}? (\langle \text{ext-tau } \tau [\mathcal{E}_1 [\tau_1 \tau_2]], (\text{ext-rho } \tau_2 \tau_3) \rangle) \Rightarrow \epsilon_v, \text{error})
\]

\textbf{induction hypothesis on } \epsilon_2

\[
= \mathcal{E} [\epsilon_1] \rho (\lambda v' \cdot \text{consis}? (\langle \text{ext-tau } \tau [\mathcal{E}_1 [\tau_1 \tau_2]], (\text{ext-rho } \tau_2 \tau_3) \rangle) \Rightarrow \epsilon_v, \text{error})
\]

\textbf{definition of } \mathcal{E}_1

\[
= \mathcal{E} [\epsilon_1] \rho (\lambda v' \cdot \text{consis}? (\langle \text{ext-tau } \tau [\mathcal{E}_1 [\tau_1 \tau_2]], (\text{ext-rho } \tau_2 \tau_3) \rangle) \Rightarrow \epsilon_v, \text{error})
\]

\textbf{definition of } \mathcal{E}_1

\[
= \mathcal{E} [\epsilon_1] \rho (\lambda v' \cdot \text{consis}? (\langle \text{ext-tau } \tau [\mathcal{E}_1 [\tau_1 \tau_2]], (\text{ext-rho } \tau_2 \tau_3) \rangle) \Rightarrow \epsilon_v, \text{error})
\]

\textbf{definition of } \mathcal{E}_1

\[
= \mathcal{E} [\epsilon_1] \rho (\lambda v' \cdot \text{consis}? (\langle \text{ext-tau } \tau [\mathcal{E}_1 [\tau_1 \tau_2]], (\text{ext-rho } \tau_2 \tau_3) \rangle) \Rightarrow \epsilon_v, \text{error})
\]

\textbf{definition of } \mathcal{E}_1

\[
= \mathcal{E} [\epsilon_1] \rho (\lambda v' \cdot \text{consis}? (\langle \text{ext-tau } \tau [\mathcal{E}_1 [\tau_1 \tau_2]], (\text{ext-rho } \tau_2 \tau_3) \rangle) \Rightarrow \epsilon_v, \text{error})
\]

\textbf{definition of } \mathcal{E}_1

\[
= \mathcal{E} [\epsilon_1] \rho (\lambda v' \cdot \text{consis}? (\langle \text{ext-tau } \tau [\mathcal{E}_1 [\tau_1 \tau_2]], (\text{ext-rho } \tau_2 \tau_3) \rangle) \Rightarrow \epsilon_v, \text{error})
\]

\textbf{definition of } \mathcal{E}_1

\[
= \mathcal{E} [\epsilon_1] \rho (\lambda v' \cdot \text{consis}? (\langle \text{ext-tau } \tau [\mathcal{E}_1 [\tau_1 \tau_2]], (\text{ext-rho } \tau_2 \tau_3) \rangle) \Rightarrow \epsilon_v, \text{error})
\]
Remark. If \( \langle \text{ext-tau } x \tau'[\delta[e_2] \tau'] \rangle \), \( \text{ext-rho } x(\text{ext-rho } ipu')v_2 \rangle \) are type consistent, then so are \( \langle \text{ext-tau } x \tau'[\delta[e_2] \tau'] \rangle \), \( \text{ext-rho } xpv_2 \rangle \). Therefore,

\[
\begin{align*}
\delta[e] \rho(\lambda v \cdot \text{consis}?) \langle \text{ext-tau } x \tau'[\delta[e] \tau'] \rangle, \ (\text{ext-rho } xpv) &\Rightarrow ev, \ \text{error} \\
= \delta[e_1] \rho(\lambda v' \cdot \text{consis}?) \langle \text{ext-tau } x \tau'[\delta[e_1] \tau'] \rangle, \ (\text{ext-rho } xpv') \rangle \\
\Rightarrow \delta[e_2][(\text{ext-rho } ipu') \\
\times (\lambda v \cdot \text{consis}?) \langle \text{ext-tau } x \tau'[\delta[e_2] \tau'] \rangle, \\
(\text{ext-rho } x(\text{ext-rho } xpv')v) \rangle \\
\Rightarrow ev_2, \ \text{error} \\
\end{align*}
\]

Therefore,

\[
\begin{align*}
\delta[e_1] \rho(\lambda v' \cdot \text{consis}?) \langle \text{ext-tau } x \tau'[\delta[e_1] \tau'] \rangle, \ (\text{ext-rho } xpv') \rangle \\
\Rightarrow \delta[e_2][(\text{ext-rho } ipu')(\lambda v_2 \cdot ev_2) \\
\text{induction hypothesis on } e_2, \ \text{again} \\
= \delta[e_1] \rho(\lambda v' \cdot \delta[e_2](\text{ext-rho } ipu')(\lambda v_2 \cdot ev_2) \\
\text{induction hypothesis on } e_1, \ \text{again} \\
= \delta[e_1] \rho(\lambda v' \cdot \delta[e_2](\text{ext-rho } ipu')v) \eta \text{ conversion} \\
= \delta[e] \rho \epsilon \text{ definition of } \delta
\end{align*}
\]

(iv) The proof for addition expressions is straightforward but long. It follows from applications of the induction hypothesis, definition of \( \text{int-add} \), and definition of \( \delta[e_1 + e_2] \).

Proof of Theorem 1.2. Let \( \langle \tau, \rho \rangle \) be type consistent. Let \( e_1, e_2 \in \text{Exp} \) and \( i \in \text{Ide} \).

(i) "Let" expression. Assume \( \delta[\text{let } i = e_1 \text{ in } e_2] \tau \neq \text{"untyped"}. \) Let \( \tau' = (\text{ext-tau } it[\delta[e_1] \tau']) \). By definition, \( \delta[e_1] \tau \neq \text{"untyped"} \) and \( \delta[e_2] \tau' \neq \text{"untyped"}. \)

\[
\begin{align*}
\delta[\text{let } i = e_1 \text{ in } e_2] \rho(\lambda v \cdot \text{isInt}? v : f_1 v \\
\text{isReal}? v : f_2 v \\
\text{isBool}? v : f_3 v) \\
= \delta[e_1] \rho(\lambda v_1 \cdot \delta[e_2](\text{ext-rho } ipu') \\
(\lambda v \cdot \text{isInt}? v : f_1 v \\
\text{isReal}? v : f_2 v \\
\text{isBool}? v : f_3 v)) \text{ definition of } \delta \\
= \delta[e_1] \rho(\lambda v_1 \cdot \text{consis}? \langle \tau', (\text{ext-rho } ipu') \rangle \\
\Rightarrow \delta[e_2](\text{ext-rho } itv)(\lambda v \cdot \text{isInt}? v : f_1 v \\
\text{isReal}? v : f_2 v \\
\text{isBool}? v : f_3 v)) \text{ Lemma 1.1 on } e_1
\end{align*}
\]


CORRECT STATIC ANALYSIS

\[ \delta[e_1] \rho(\lambda v_1 \cdot \text{cons? } \langle \tau', (\text{ext-rho itv}) \rangle) = \delta[e_2](\text{ext-rho itv})(\lambda v \cdot \delta[e_2] v) \tau' = \text{"int" } : f_1 v \]

\[ \delta[e_2] v \tau' = \text{"real" } : f_2 v \]

\[ \delta[e_2] v \tau' = \text{"bool" } : f_3 v, f_4 v) \]

**induction hypothesis on** \( e_2 \)

\[ \delta[e_1] \rho(\lambda v_1 \cdot \delta[e_2] v)(\text{ext-tau itv})(\lambda v \cdot \delta[e_2] v) \tau' = \text{"int" } : f_1 v \]

\[ \delta[e_2] v \tau' = \text{"real" } : f_2 v \]

\[ \delta[e_2] v \tau' = \text{"bool" } : f_3 v, f_4 v) \]

**Lemma 1.1, again**

\[ \delta[e_1] \rho(\lambda v_1 \cdot \delta[e_2] v)(\text{ext-tau itv})(\lambda v \cdot \delta[e_2] v) \tau' = \text{"int" } : f_1 v \]

\[ \delta[e_2] v \tau' = \text{"real" } : f_2 v \]

\[ \delta[e_2] v \tau' = \text{"bool" } : f_3 v, f_4 v) \]

**definition of** \( \delta \)

\[ \delta[\text{let } i = e_1 \text{ in } e_2] \rho(\lambda v_2 \cdot \delta[\text{let } i = e_1 \text{ in } e_2] v) \tau' = \text{"int" } \Rightarrow f_1 v \]

\[ \delta[e_2] v \tau' = \text{"real" } \Rightarrow f_2 v \]

\[ \delta[e_2] v \tau' = \text{"bool" } \Rightarrow f_3 v, f_4 v) \]

**definition of** \( \delta \).

(ii) "Addition" expression. Assume \( \delta[(e_1 + e_2)] v \neq \text{"untyped"}. By definition, \( \delta[e_1] v \neq \text{"untyped"} \) and \( \delta[e_2] v \neq \text{"untyped"}."

\[ \delta[(e_1 + e_2)] v = \delta[e_1] v \rho(\lambda v_1 \cdot \text{isInt? } v_1 : \delta[e_2] v \rho(\lambda v_2 \cdot \text{isInt? } v_2 : \varepsilon[((v_1 | \text{Int}) + (v_2 | \text{Int})) \text{ inEvaluate}] \text{isReal? } v_2 : \varepsilon[((v_1 | \text{Int}) \text{ inReal } | \text{Real}) + (v_2 | \text{Real})) \text{ inEvaluate}] \text{isBool? } v_2 : \text{terminate error}_2) \]

\[ \text{isReal? } v_1 : \delta[e_2] v \rho(\lambda v_2 \cdot \text{isInt? } v_2 : \varepsilon[((v_1 | \text{Real}) + (v_2 | \text{Int}) \text{ inReal } | \text{Real})) \text{ inEvaluate}] \text{isReal? } v_2 : \varepsilon[((v_1 | \text{Real}) + (v_2 | \text{Real})) \text{ inEvaluate}] \text{isBool? } v_2 : \text{terminate error}_2) \]

\[ \text{isBool? } v_1 : \delta[e_2] v \rho(\lambda v_2 \cdot \text{isInt? } v_2 : \text{terminate error}_2 \text{isReal? } v_2 : \text{terminate error}_2 \text{isBool? } v_2 : \varepsilon[((v_1 | \text{Bool}) + (v_2 | \text{Bool})) \text{ inEvaluate}]))) \]

**definition of** \( \delta \)

\[ \delta[e_1] v \rho(\lambda v_1 \cdot \text{isInt? } v_1 : \delta[e_2] v \rho(\lambda v_2 \cdot \text{isInt? } v_2 : \varepsilon[int-add \rho e_1 v_2] v_2) \]
isReal? \( v_2 : \text{coerce } \rho(\lambda x \cdot \text{real-add } p \varepsilon v_2) v_1 \)

isBool? \( v_2 : \text{terminate } \text{error}_2 \)

isReal? \( v_1 : \) 
\[ \delta[e_2] \rho(\lambda v_2). \]

isInt? \( v_2 : \text{coerce } \rho(\text{real-add } p \varepsilon v_1) v_2 \)

isReal? \( v_2 : \text{real-add } p \varepsilon v_1 v_2 \)

isBool? \( v_2 : \text{terminate } \text{error}_2 \)

isBool? \( v_1 : \) 
\[ \delta[e_2] \rho(\lambda v_2). \]

isInt? \( v_2 : \text{terminate } \text{error}_2 \)

isReal? \( v_2 : \text{terminate } \text{error}_2 \)

isBool? \( v_2 : \text{bool-or } p \varepsilon v_1 v_2 ) \)

\[ = \delta[e_1] \rho(\lambda v_1). \]

\[ \delta[e_1] \tau = \text{"int" } \Rightarrow \]
\[ \delta[e_2] \rho(\lambda v_2). \]

\[ \delta[e_1] \tau = \text{"real" } \Rightarrow \text{coerce } \rho(\lambda x \cdot \text{real-add } p \varepsilon v_2) v_1 \]

\[ \delta[e_2] \tau = \text{"bool" } \Rightarrow \text{terminate } \text{error}_2 \]

\[ = \delta[e_1] \rho(\lambda v_1). \]

\[ \delta[e_2] \tau = \text{"int" } \Rightarrow \text{coerce } \rho(\text{real-add } p \varepsilon v_1) v_2 \]

\[ \delta[e_2] \tau = \text{"real" } \Rightarrow \text{real-add } p \varepsilon v_1 v_2 \]

\[ \delta[e_2] \tau = \text{"bool" } \Rightarrow \text{terminate } \text{error}_2 \]

\[ \delta[e_1] \tau = \text{"bool" } \Rightarrow \]
\[ \delta[e_2] \rho(\lambda v_2). \]

\[ \delta[e_2] \tau = \text{"int" } \Rightarrow \text{terminate } \text{error}_2 \]

\[ \delta[e_2] \tau = \text{"real" } \Rightarrow \text{terminate } \text{error}_2 \]

\[ \delta[e_2] \tau = \text{"bool" } \Rightarrow \text{bool-or } p \varepsilon v_1 v_2 \]

\[ \delta[e_2] \tau = \text{"bool" } \Rightarrow \text{terminate } \text{error}_2 \]

induction hypothesis on \( e_1 \) and \( e_2 \).

Call the right side of the above expression \( \alpha \). We must show \( \alpha[x/e] = \alpha[y/e] \), where

\[ x = (\lambda v \cdot \text{isInt? } v : f_1 v \]

\[ \text{isReal? } v : f_2 v \]

\[ \text{isBool? } v : f_3 v \]

and

\[ y = (\lambda v \cdot \delta[e_1 + e_2] \tau = \text{"int" } \Rightarrow f_1 v \]

\[ \delta[e_1 + e_2] \tau = \text{"real" } \Rightarrow f_2 v \]

\[ \delta[e_1 + e_2] \tau = \text{"bool" } \Rightarrow f_3 v \]

\[ , f_2 v \).\]
Henceforth, the above expression will be written as

\[ \alpha([\lambda v \cdot \text{isInt? } v : f_1 v \\
\text{isReal? } v : f_2 v \\
\text{isBool? } v : f_3 v]/\varepsilon) = \alpha([\lambda v \cdot \varepsilon[[e_1 + e_2]]] \tau = \text{"int" } \Rightarrow f_1 v \\
\varepsilon[[e_1 + e_2]] \tau = \text{"real" } \Rightarrow f_2 v \\
\varepsilon[[e_1 + e_2]] \tau = \text{"bool" } \Rightarrow f_3 v)
\]

Case. \( \varepsilon[[e_1]] \tau = \text{"int."} \)

Subcase. \( \varepsilon[[e_2]] \tau = \text{"int,"} \) so \( \varepsilon[[e_1 + e_2]] \tau = \text{"int"} \) therefore,

\[ \alpha([\lambda v \cdot \varepsilon[[e_1 + e_2]]] \tau = \text{"int" } \Rightarrow f_1 v \\
\varepsilon[[e_1 + e_2]] \tau = \text{"real" } \Rightarrow f_2 v \\
\varepsilon[[e_1 + e_2]] \tau = \text{"bool" } \Rightarrow f_3 v \\
\]

The cases for \( \varepsilon[[e_2]] \tau = \text{"real"} \) and \( \varepsilon[[e_2]] \tau = \text{"bool"} \) are similar to the subcase above.

Proof of Theorem 2.3. Let \( \langle \tau, \rho \rangle \) be type consistent.

(i) \( e \in \langle \text{Const} \rangle. \) This case is a direct result of the definitions of \( \varepsilon, \varepsilon', \) and \( \varepsilon'. \)

(ii) \( e ::= i. \) Assume \( \varepsilon[[i]] \tau \neq \text{"untyped."} \) By definition \( \tau i \neq \text{"untyped."} \)

By definition,

\[ \varepsilon[[i]] = \lambda \rho e \cdot \rho i = \text{"undeclared" } \Rightarrow \text{terminate error}, \]

\[ , \varepsilon[\rho i] \text{ Evalue}. \]
The proof proceeds by cases on $\rho i \in \text{Dvalue}$. By consistency, $\rho i \neq \bot_{\text{Dvalue}}$ and $[\rho i | \text{Eval}] \neq \bot_{\text{Eval}}$. By assumption, $\tau i \neq \"\text{untyped}\"$, so by consistency, $\rho i \neq \"\text{undeclared}\"$.

Case. isInt?($\rho i$) implies, by consistency, $\tau i = \"\text{int}\"$ therefore by definition,

$$\delta'[i] \tau p e = \delta'[i] \rho e = e[\rho i | \text{Eval}] .$$

The cases for isReal? ($\rho i$) and isBool? ($\rho i$) are similar.

(iii) Let $e = \text{let } i = e_1 \text{ in } e_2$. Assume $\delta'[e] \tau \neq \"\text{untyped}\"$. Let $\tau' = (\text{ext-tau } it[\delta'[e_1] \tau])$. By definition, $\delta'[e_1] \tau \neq \"\text{untyped}\"$ and $\delta'[e_2] \tau' \neq \"\text{untyped}\"$.

$$\delta'[e] \tau p e = \delta'[e_1] \tau p (\lambda v_1 \cdot \delta'[e_2] \tau' (\text{ext-rho } iv_1)) e \quad \text{definition of } \delta'$$

$$= \delta'[e_1] \rho (\lambda v_1 \cdot \delta'[e_2]) \tau' (\text{ext-rho } iv_1) e \quad \text{induction hypothesis on } e_1$$

$$= \delta'[e_1] \rho (\lambda v_1 \cdot \text{consis?}) (\text{ext-rho } iv_1)) e$$

$$\Rightarrow \delta'[e_2] \tau' (\text{ext-rho } ipv_1) e \quad \text{Lemma 1.1 on } e_1$$

$$= \delta'[e_1] \rho (\lambda v_1 \cdot \text{consis?}) (\text{ext-rho } ipv_1)) e$$

$$\Rightarrow \delta'[e_2] (\text{ext-rho } ipv_1) e \quad \text{induction hypothesis on } e_2$$

$$= \delta'[e] \rho e \quad \text{definition of } \delta$$

(iv) Let $e = (e_1 + e_2)$.

$$\delta[(e_1 + e_2)]$$

$$= \lambda p e \cdot \delta[e_1] \rho (\lambda v_1 \cdot$$

isInt? $v_1$:

$$\delta[e_2] \rho (\lambda v_2 \cdot$$

isInt? $v_2$:

$$\epsilon[((v_1 | \text{Int}) + (v_2 | \text{Int})) \text{ inEval}]}$$

isReal? $v_2$:

$$\epsilon[((v_1 | \text{Int}) \text{ inEval}]) (\text{Real}) + (v_2 | \text{Real}) \text{ inEval}]}$$

isBool? $v_2$:

$$\epsilon[(\text{terminate error}_2)]$$

isBool? $v_1$:

$$\delta[e_2] \rho (\lambda v_2 \cdot$$

isInt? $v_2$:

$$\epsilon[((v_1 | \text{Real}) + (((v_2 | \text{Real}) \text{ inEval}])) \text{ inEval}]}$$

isReal? $v_2$:

$$\epsilon[((v_1 | \text{Real}) + (v_2 | \text{Real}) \text{ inEval}]}$$

isBool? $v_2$:

$$\epsilon[(\text{terminate error}_2)]$$

isBool? $v_1$:

$$\delta[e_2] \rho (\lambda v_2 \cdot$$

isInt? $v_2$:

$$\epsilon[(\text{terminate error}_2)]$$

isReal? $v_2$:

$$\epsilon[(\text{terminate error}_2)]$$

isBool? $v_2$:

$$\epsilon[((v_1 | \text{Bool}) + (v_2 | \text{Bool}) \text{ inEval}]}$$

$$\text{definition of } \delta$$
CORRECT STATIC ANALYSIS

\[ = \lambda \rho \cdot \mathcal{E}[e_1] \cdot \rho(\lambda v_1) \]

**isInt?** \(v_1\):
\[ \mathcal{E}[e_2] \cdot \rho(\lambda v_2) \]
**isInt?** \(v_2\) : \( \text{int-add } \rho(\lambda v_1, v_2) \)
**isReal?** \(v_2\) : \( \text{coerce } \rho(\lambda x \cdot \text{real-add } \rho(\varepsilon x v_2))v_1 \)
**isBool?** \(v_2\) : \( \text{terminate error}_2 \)

**isReal?** \(v_1\) :
\[ \mathcal{E}[e_2] \cdot \rho(\lambda v_2) \]
**isInt?** \(v_2\) : \( \text{coerce } \rho(\text{real-add } \rho(\varepsilon v_1))v_1 \)
**isReal?** \(v_2\) : \( \text{real-add } \rho(\varepsilon v_1) \)
**isBool?** \(v_2\) : \( \text{terminate error}_2 \)

**isBool?** \(v_1\) :
\[ \mathcal{E}[e_2] \cdot \rho(\lambda v_2) \]
**isInt?** \(v_2\) : \( \text{terminate error}_2 \)
**isReal?** \(v_2\) : \( \text{terminate error}_2 \)
**isBool?** \(v_2\) : \( \text{bool-or } \rho(\varepsilon v_1, v_2) \)

**definition of auxiliaries**

\[ = \lambda \rho \cdot \mathcal{E}[e_1] \cdot \rho(\lambda v_1) \]
**\(\mathcal{E},[e_1]\)\(\tau\) = “int”:
\[ \mathcal{E}[e_2] \cdot \rho(\lambda v_2) \]
**\(\mathcal{E},[e_2]\)\(\tau\) = “int” : \( \text{int-add } \rho(\varepsilon v_1, v_2) \)
\[ \mathcal{E}[e_2] \cdot \rho(\lambda v_2) \]
**\(\mathcal{E},[e_2]\)\(\tau\) = “real” : \( \text{coerce } \rho(\lambda x \cdot \text{real-add } \varepsilon x v_2) \)
**\(\mathcal{E},[e_2]\)\(\tau\) = “bool” : \( \text{terminate error}_2 \)

**\(\mathcal{E},[e_1]\)\(\tau\) = “real”:
\[ \mathcal{E}[e_2] \cdot \rho(\lambda v_2) \]
**\(\mathcal{E},[e_2]\)\(\tau\) = “int” : \( \text{coerce } \rho(\text{real-add } \varepsilon v_1) \)
**\(\mathcal{E},[e_2]\)\(\tau\) = “real” : \( \text{real-add } \varepsilon v_1 \)
**\(\mathcal{E},[e_2]\)\(\tau\) = “bool” : \( \text{terminate error}_2 \)

isBool? \(v_1\) :
\[ \mathcal{E}[e_2] \cdot \rho(\lambda v_2) \]
**\(\mathcal{E},[e_2]\)\(\tau\) = “int” : \( \text{terminate error}_2 \)
**\(\mathcal{E},[e_2]\)\(\tau\) = “real” : \( \text{terminate error}_2 \)
**\(\mathcal{E},[e_2]\)\(\tau\) = “bool” : \( \text{bool-or } \rho(\varepsilon v_1, v_2) \) 

**Theorem 1.2**

\[ = \lambda \rho \cdot \mathcal{E}[e_1] \cdot \rho(\lambda v_1) \]
**“int”** case \(\mathcal{E},[e_2]\)(\(\tau\)) of

\[ \mathcal{E}[e_1] \cdot \rho(\lambda v_1) \]
**“int”** : \( \mathcal{E}[e_2] \cdot \rho(\lambda v_2) \cdot \rho(\lambda v_1 \cdot \text{int-add } \varepsilon v_1, v_2) \)
**“real”** : \( \mathcal{E}[e_1] \cdot \rho(\lambda v_1) \cdot \rho(\lambda v_2 \cdot \text{coerce } \rho(\lambda x \cdot \text{real-add } \varepsilon x v_2) v_1) \)
**“bool”** : \( \mathcal{E}[e_1] \cdot \rho(\lambda v_1) \cdot \rho(\lambda v_2 \cdot \text{terminate error}_2) \)
**“untyped”** : \( \mathcal{E}[e_1] \cdot \rho(\lambda v_1) \cdot \rho(\lambda v_2 \cdot \text{terminate error}_3) \)
**“real”** case \(\mathcal{E}[e_2]\)(\(\tau\)) of

\[ \mathcal{E}[e_1] \cdot \rho(\lambda v_1) \]
**“int”** : \( \mathcal{E}[e_2] \cdot \rho(\lambda v_2 \cdot \text{coerce } \rho(\text{real-add } \varepsilon v_1) v_2) \)
**“real”** : \( \mathcal{E}[e_2] \cdot \rho(\lambda v_2 \cdot \text{real-add } \varepsilon v_1, v_2) \)
"bool" : $\lambda [e_1 : e_2] \rho (\lambda v_1 : \delta [e_2] \rho (\lambda v_2 : \text{terminate error}_2))$

"untyped": $\delta [e_1] \rho (\lambda v_1 : \delta [e_2] \rho (\lambda v_2 : \text{terminate error}_5))$

"bool": case $\delta [e_2] \tau$

"int" : $\delta [e_1] \rho (\lambda v_1 : \delta [e_2] \rho (\lambda v_2 : \text{terminate error}_2))$

"real" : $\delta [e_1] \rho (\lambda v_1 : \delta [e_2] \rho (\lambda v_2 : \text{terminate error}_2))$

"bool" : $\delta [e_1] \rho (\lambda v_1 : \delta [e_2] \rho (\lambda v_2 : \text{bool-or-psi}_1 v_2))$

"untyped": $\delta [e_1] \rho (\lambda v_1 : \delta [e_2] \rho (\lambda v_2 : \text{terminate error}_5))$

"untyped": $\delta [e_1] \rho (\lambda v_1 : \delta [e_2] \rho (\lambda v_2 : \text{terminate error}_5))$

Corollary 2.2

= $\delta'[(e_1 + e_2)] \tau$

induction hypothesis on $e_1$ and $e_2$

Proof of Theorem 5.4. Let $\langle \tau, \rho, \sigma \rangle$ be type consistent and let $\kappa \in \text{Ccont}$ be arbitrary. Then we claim $\mathcal{S} [\text{while } e \text{ do } s] \rho \kappa \sigma = \mathcal{S}' [\text{while } e \text{ do } s] \tau \rho \kappa \sigma$.

$\mathcal{S} [\text{while } e \text{ do } s] \tau = \text{"ok"}$ implies $\delta [e] \tau = \text{"bool"}$ and $\mathcal{S} [s] \tau = \text{"ok"}$.

Therefore

$\mathcal{S}' [\text{while } e \text{ do } s] \tau \rho \kappa \sigma$

= fix$(\lambda \theta \cdot \delta'[e] \tau \rho (\lambda v \cdot [v \mid \text{Bool}] = \text{true} \Rightarrow \mathcal{S}[s] \tau \theta, \kappa)) \sigma$.

By definition,

$\mathcal{S} [\text{while } e \text{ do } s] \rho \kappa \sigma$

= fix$(\lambda \theta \cdot \delta[e] \rho (\lambda v \cdot \text{isBool? } v \cdot [v \mid \text{Bool}] = \text{true} \Rightarrow \mathcal{S}[s] \rho \theta, \kappa$

else: $\text{terminate error}_6)) \sigma$.

Let

$f = (\lambda \theta \cdot \delta[e] \rho (\lambda v \cdot \text{isBool? } v \cdot [v \mid \text{Bool}] = \text{true} \Rightarrow \mathcal{S}[s] \rho \theta, \kappa$

else: $\text{terminate error}_6))$.

Let $g = (\lambda \theta \cdot \delta'[e] \tau \rho (\lambda v \cdot [v \mid \text{Bool}] = \text{true} \Rightarrow \mathcal{S}[s] \rho \tau \theta, \kappa)) \sigma$.

We must show that $(\text{fix } f) \sigma = (\text{fix } g) \sigma$ for all $\sigma \in \text{State}$ such that $\langle \tau, \rho, \sigma \rangle$ are type consistent. Fix $\tau$ and $\rho$ and call $\sigma$ compatible if $\langle \tau, \rho, \sigma \rangle$ are type consistent. Call $\theta, \theta' \in \text{Ccont}$ similar if $\theta \sigma = \theta' \sigma$ for all compatible $\sigma$.

Claim. If $\theta$ and $\theta'$ are similar, then so are $f \theta$ and $g \theta'$ for $\theta, \theta' \in \text{Ccont}$.

Claim. If $f \theta \sigma = \delta[e] \rho (\lambda v \cdot \text{isBool? } v \cdot [v \mid \text{Bool}] = \text{true} \Rightarrow \mathcal{S}[s] \rho \theta, \kappa$

else: $\text{terminate error}_6) \sigma$

= $\delta[e] \rho (\lambda v \cdot \text{"bool"} \cdot [v \mid \text{Bool}] = \text{true} \Rightarrow \mathcal{S}[s] \rho \theta, \kappa$

else: $\text{terminate error}_6) \sigma$

Theorem 5.3

= $\delta[e] \rho (\lambda v \cdot \text{true} \Rightarrow \mathcal{S}[s] \rho \theta, \kappa) \sigma$ $\delta[e] \tau = \text{"bool"}$

= $\delta[e] \rho (\lambda v \cdot \text{consis? } \langle \tau, \rho, \sigma' \rangle \Rightarrow [v \mid \text{Bool}] = \text{true} \Rightarrow \mathcal{S}[s] \rho \theta \sigma', \kappa \sigma'$

Lemma 5.2 on $e$
\[ \delta[e] \rho(\lambda v \sigma' \cdot \text{consis?} \langle \tau, \rho, \sigma' \rangle) \]
\[ \Rightarrow [v \mid \text{Bool}] = \text{true} \]
\[ \Rightarrow \mathcal{S}[s] \rho(\lambda \sigma'' \cdot \text{consis?} \langle \tau, \rho, \sigma'' \rangle \Rightarrow \theta \sigma'', \text{error}) \sigma' \]
\[ \Rightarrow [t \mid \text{Bool}] = \text{true} \]
\[ \Rightarrow \mathcal{S}[s] \rho(\lambda \sigma'' \cdot \text{consis?} \langle \tau, \rho, \sigma'' \rangle \Rightarrow \theta' \sigma'', \text{error}) \sigma' \]
\[ \Rightarrow \text{error} \]
\[ \text{Lemma 5.2 on } s \]
\[ \delta[e] \rho(\lambda v \sigma' \cdot \text{consis?} \langle \tau, \rho, \sigma' \rangle) \]
\[ \Rightarrow [v \mid \text{Bool}] = \text{true} \]
\[ \Rightarrow \mathcal{S}[s] \rho(\lambda \sigma'' \cdot \text{consis?} \langle \tau, \rho, \sigma'' \rangle \Rightarrow \theta \sigma'', \text{error}) \sigma' \]
\[ \Rightarrow [t \mid \text{Bool}] = \text{true} \]
\[ \Rightarrow \mathcal{S}[s] \rho(\lambda \sigma'' \cdot \text{consis?} \langle \tau, \rho, \sigma'' \rangle \Rightarrow \theta' \sigma'', \text{error}) \sigma' \]
\[ \Rightarrow \text{error} \]
\[ \text{Lemma 5.2 on } s, \text{ again} \]
\[ \delta[e] \rho(\lambda v \sigma' \cdot \text{consis?} \langle \tau, \rho, \sigma' \rangle) \]
\[ \Rightarrow [v \mid \text{Bool}] = \text{true} \Rightarrow \mathcal{S}[s] \rho \theta' \sigma', \kappa \sigma' \]
\[ \Rightarrow [t \mid \text{Bool}] = \text{true} \Rightarrow \mathcal{S}[s] \rho \theta' \sigma', \kappa \sigma' \]
\[ \Rightarrow \text{error} \]
\[ \text{induction hypothesis on } s \]
\[ \delta[e] \rho(\lambda v \sigma' \cdot [v \mid \text{Bool}] = \text{true} \Rightarrow \mathcal{S}[s] \rho \theta' \sigma', \kappa \sigma' \sigma) \]
\[ \Rightarrow \text{Lemma 5.2 on } e, \text{ again} \]
\[ \delta'[e] \rho \theta(\lambda v \sigma' \cdot [v \mid \text{Bool}] = \text{true} \Rightarrow \mathcal{S}'[s] \rho \theta' \sigma', \kappa \sigma' \sigma) \]
\[ \Rightarrow \text{induction hypothesis on } e \]
\[ \delta'[e] \rho \theta(\lambda v \cdot [v \mid \text{Bool}] = \text{true} \Rightarrow \mathcal{S}'[s] \rho \theta' \sigma', \kappa \sigma) \]
\[ \Rightarrow \eta \text{ conversion} \]
\[ \Rightarrow g \theta' \sigma. \]

Therefore, \( f \theta \) is similar to \( g \theta' \). Hence, for all \( n \), \( f^n(\bot) \) and \( g^n(\bot) \) are similar.

Now \( (\text{fix } f) \sigma = \bigcup \bot, f \bot \sigma, f(f \bot) \sigma, \ldots \), and \( (\text{fix } g) \sigma = \bigcup \bot, g \bot \sigma, g(g \bot) \sigma, \ldots \), and these approximations are pairwise equal because \( \sigma \) is compatible. Therefore, \( (\text{fix } f) \sigma = (\text{fix } g) \sigma \) for all \( \sigma \in \text{State} \) such that \( \langle \tau, \rho, \sigma \rangle \) are type consistent.

Received September 22, 1987; accepted July 21, 1988

REFERENCES


