Program Transformation via Contextual Assertions

Ian A. Mason and Carolyn Talcott

Stanford University, Stanford, California, 94305-2140, USA

Abstract. In this paper we describe progress towards a theory of tranformational program development. The transformation rules are based on a theory of contextual equivalence for functional languages with imperative features. Such notions of equivalence are fundamental for the process of program specification, derivation, transformation, refinement and other forms of code generation and optimization. This paper is dedicated to Professor Satoru Takasu.

1 Introduction

This paper describes progress towards a theory of program development by systematic refinement beginning with a clean simple program thought of as a specification. Transformations include reuse of storage, and re-representation of abstract data. The transformation rules are based on a theory of constrained equivalence for functional languages with imperative features (i.e. Lisp, Scheme or ML). Such notions of equivalence are fundamental for the process of program specification, derivation, transformation, refinement, and other forms of code generation and optimization. This paper is a continuation of our development of the Variable Typed Logic of Effects (VTLoE) introduced in [11, 20, 10]. VTLoE is inspired by the variable type systems of Feferman. These systems are two sorted theories of operations and classes initially developed for the formalization of constructive mathematics [2, 3] and later applied to the study of purely functional languages [4, 5]. VTLoE goes well beyond traditional programming logics, such as Hoare's [1] and Dynamic logic [9]. The programming language and logic are richer. It is close in spirit to Specification Logic [27], incorporating a full first order theory of data and the ability to express program equivalence, and to assert and nest Hoare-like triples (called *contextual assertions* in VTLoE). The underlying programming languages are quite different: Specification Logic concerns Algol-like programs that are strongly typed, can store only first-order data, and obey call-by-name semantics; while VTLoE concerns untyped ML- or Scheme-like languages that can store arbitrary values, and obey call-by-value semantics. The underlying programming language of VTLoE, $\lambda_{\mathtt{mk}}$, is based on the call-by-value lambda calculus extended by the reference primitives `mk`, `set`, `get`. The cornerstone of VTLoE is the notion of operational equivalence. Two expressions are operationally equivalent if they cannot be distinguished by any program context. Operational equivalence enjoys many nice properties such as being a

congruence relation on expressions. It subsumes the lambda-v-calculus [26] and the lambda-c calculus [22]. The theory of operational equivalence for $\lambda_{\mathtt{mk}}$ is presented in [17, 18].

In [10] we began the development of the full logic of VTLoE, including general properties of contextual assertions and valid forms of class comprehension. In this setting induction principles for reasoning about programs can derived using minimal and maximal fixed-points of class operators. Another important semantic technique for establishing properties of programs is induction on the length of computations. In this paper we focus on formalizing principles for program transformation based on computation induction. Rather than deal with the full $\lambda_{\mathtt{mk}}$ calculus, we restrict our attention to the first order fragment (where there are no higher-order procedures). This simplifies the presentation substantially. For this fragment we present four main tools: subgoal induction, recursion induction, peephole optimization, and a memory reuse principle. The subgoal and recursion induction principles generalize standard principles for purely functional languages. Subgoal induction reduces proving input-output assertions about recursively defined programs to finding suitable invariants and showing that they propagate across the functions defining bodies. Recursion induction reduces proving equivalence of two recursive programs to showing that each satisfies the others defining equation. The peephole rule answers affirmatively a conjecture in [13], p 178. This rule allows for the replacement of a program fragment with another fragment that is equivalent in the context of use. The memory-reuse principle captures many of the standard compiler optimizations concerning register and memory reuse. This principle strengthens recursion induction to account for local memory.

We give two examples of the use of these tools. The first is the transformation of a tail recursive program to a loop that utilizes a register. The second is the transformation of a simple, but non-trivial, rewriting program, known as the Boyer benchmark [8]. This example is part of a more extensive transformational development carried out in the process of developing parallel Lisp programs for symbolic manipulation [15].

The remainder of this paper is organized as follows. In section 2 we present the syntax and semantics of terms. In section 3 we describe the first-order fragment of VTLoE. In section 4 we present the main transformation tools. In section 5 these tools are used to transform the Boyer Benchmark. Section 6 contains concluding remarks and directions for future work.

We conclude this section with a summary of notational conventions. We use the usual notation for set membership and function application. Let X, Y, Y_0, Y_1 be sets. We specify meta-variable conventions in the form: let x range over X, which should be read as: the meta-variable x and decorated variants such as x', x_0, ..., range over the set X. Y^n is the set of sequences of elements of Y of length n. Y^* is the set of finite sequences of elements of Y. $[y_1, \ldots, y_n]$ is the sequence of length n with ith element y_i. $[Y_0 \to Y_1]$ is the set of functions f with domain Y_0 and range contained in Y_1. We write $\mathrm{Dom}(f)$ for the domain of a function and $\mathrm{Rng}(f)$ for its range. $\mathrm{Fmap}[Y_0, Y_1]$ is the set of functions whose

domain is a *finite* subset of Y_0 and whose range is a subset of Y_1. For any function f, $f\{y := y'\}$ is the function f' such that $\mathrm{Dom}(f') = \mathrm{Dom}(f) \cup \{y\}$, $f'(y) = y'$, and $f'(z) = f(z)$ for $z \neq y, z \in \mathrm{Dom}(f)$. $\mathbb{N} = \{0, 1, 2, \ldots\}$ is the set of natural numbers and $i, j, n, n_0, \ldots$ range over $\mathbb{N}$.

2 The Syntax and Semantics of Terms

Our language can be thought of as a first-order untyped ML, or as a variant of Scheme in which naming of values and memory allocation have been separated. Thus there are explicit memory operations (`atom?`, `cell?`, `eq?`, `mk`, `get`, `set`) but no assignment to bound variables. The reason for the choice is that it simplifies the semantics and allows one to separate the functional aspects from the imperative ones in a clean way. We also include various forms of structured data such as numbers and immutable pairs. The presentation in this section is by necessity terse. A more detailed presentation can be found in [19].

2.1 The Syntax of Terms

We fix a countably infinite set of atoms, $\mathbb{A}$, with two distinct elements playing the role of booleans, `T` for *true* and `Nil` for *false*. We also fix a countable set $\mathbb{X}$ of variables and for each $n \in \mathbb{N}$ a countable set, $\mathbb{F}_n$, of n-ary function symbols. We assume the sets $\mathbb{A}$, $\mathbb{X}$, and $\mathbb{F}_n$ for $n \in \mathbb{N}$ are pairwise disjoint. We let $\mathbb{O}_n$ denote the set of n-ary operations, $\mathbf{F}_n$ denote $\mathbb{O}_n \cup \mathbb{F}_n$ and $\mathbf{F}$ denote $\bigcup_{n \in \mathbb{N}} \mathbf{F}_n$.

The operations, $\mathbb{O}$, are partitioned into memory operations and operations that are independent of memory. A memory operation may modify memory, and its result may depend on state of memory when it is executed. The memory operations are:

$$\{\mathtt{get}, \mathtt{mk}\} \subseteq \mathbb{O}_1 \quad \{\mathtt{set}\} \subseteq \mathbb{O}_2.$$

The remaining operations neither affect the memory, nor are affected by the memory. In this paper we explicitly include operations for dealing with immutable pairs and assorted predicates.

$$\{\mathtt{cell?}, \mathtt{atom?}, \mathtt{nat?}, \mathtt{fst}, \mathtt{snd}, \mathtt{pr?}\} \subseteq \mathbb{O}_1 \qquad \{\mathtt{eq?}, \mathtt{pr}\} \subseteq \mathbb{O}_2$$

In addition there are operations on numbers which we shall not enumerate explicitly.

Definition ($\mathbb{V}$ $\mathbb{P}$ $\mathbb{S}$ $\mathbb{E}$): The set of value expressions, $\mathbb{V}$, the set of immutable pairs, $\mathbb{P}$, the set of value substitutions, $\mathbb{S}$, and the set of expressions, $\mathbb{E}$, are defined, mutually recursively, as the least sets satisfying the following equations:

$$\mathbb{V} = \mathbb{X} + \mathbb{A} + \mathbb{P}$$

$$\mathbb{P} = \mathbf{pr}(\mathbb{V}, \mathbb{V})$$

$$\mathbb{S} = \mathbf{Fmap}[\mathbb{X}, \mathbb{V}]$$

$$\mathbb{E} = \mathbb{V} \cup \mathtt{let}\{\mathbb{X} := \mathbb{E}\}\mathbb{E} \cup \mathtt{if}(\mathbb{E}, \mathbb{E}, \mathbb{E}) \cup \bigcup_{n \in \mathbb{N}} \mathbf{F}_n(\mathbb{E}^n)$$

We let a range over $\mathbb{A}$, x range over $\mathbb{X}$, v range over $\mathbb{V}$, f, g range over $\mathbb{F}$, σ range over $\mathbb{S}$, ϑ range over $\mathbb{O}$, and e range over $\mathbb{E}$. The variable of a `let` is bound in the second expression, and the usual conventions concerning alpha conversion apply. We write $\mathrm{FV}(e)$ for the set of free variables of e. $e^{\{x:=e'\}}$ is the result of substituting e' for x in e taking care not to trap free variables of e'. e^{σ} is the result of simultaneously substituting free occurrences of $x \in \mathrm{Dom}(\sigma)$ in e by $\sigma(x)$, again taking care not to trap free variables. For any syntactic domain Y and set of variables X we let Y_X be the elements of Y with free variables in X. A *closed expression* is an expression with no free variables. Thus $\mathbb{E}_{\emptyset}$ is the set of all closed expressions.

Definition ($\mathbb{C}$): Contexts are expressions with holes. We use $\bullet$ to denote a hole. The set of contexts, $\mathbb{C}$, is defined by

$$\mathbb{C} = \{\bullet\} + \mathbb{X} + \mathbb{A} + \mathtt{let}\{\mathbb{X} := \mathbb{C}\}\mathbb{C} + \mathtt{if}(\mathbb{C}, \mathbb{C}, \mathbb{C}) + \mathbf{F}_n(\mathbb{C}^n)$$

We let C range over $\mathbb{C}$. $C[e]$ denotes the result of replacing any hole in C by e. Free variables of e may become bound in this process.

Definition (Δ): The set of (recursive function-) definition systems is the collection of all finite sequences whose elements are of the form $f(x_1, \ldots, x_n) \leftarrow e$ where f is an n-ary function symbol. In symbols:

$$\Big(\bigcup_{n \in \mathbb{N}} \mathbb{F}_n(\mathbb{X}^n) \leftarrow \mathbb{E}\Big)^*$$

Let Δ be a definition system. The defined functions of Δ are those $f \in \mathbb{F}$ for which there is a definition $f(x_1, \ldots, x_n) \leftarrow e$ occurring in Δ for some $x_1, \ldots, x_n$, and e. The variables $(x_1, \ldots, x_n)$ are called the formal parameters of the definition and e is the body. A definition system Δ is well-formed if no function symbol is defined more than once, and if for each $f(x_1, \ldots, x_n) \leftarrow e$ in Δ, the variables $x_1, \ldots, x_n$ are distinct, $\mathrm{FV}(e) \subseteq \{x_1, \ldots, x_n\}$, and the function symbols occurring in e are among the defined functions of Δ. We shall assume that definition systems are well-formed unless otherwise stated. Within a single definition the formal parameters are bound in the body and we may freely α-convert (subject to maintaining well-formedness).

In order to make programs easier to read, we introduce some abbreviations.

$\mathtt{seq}(e)$	abbreviates	e	
$\mathtt{seq}(e_0, \ldots, e_n)$	abbreviates	$\mathtt{let}\{d := e_0\}\mathtt{seq}(e_1, \ldots, e_n)$	d fresh
$\mathtt{cond}()$	abbreviates	$\mathtt{Nil}$	
$\mathtt{cond}([e_0 \rhd e_0'], [e_1 \rhd e_1'], \ldots, [e_n \rhd e_n'])$	abbreviates	$\mathtt{if}(e_0, e_0', \mathtt{cond}([e_1 \rhd e_1'], \ldots, [e_n \rhd e_n']))$	

2.2 The Semantics of Terms

The operational semantics of expressions relative to a definition set Δ is given by a reduction relation $\overset{*}{\mapsto}$ on expressions. Computation is a process of stepwise reduction of an expression to a canonical form. In order to define the reduction rules we introduce the notions of *memory context*, *reduction context*, and *redex*. Redexes describe the primitive computation steps. A primitive step is either a `let`-reduction, branching according to whether a test value is `Nil` or not, unfolding an application of a recursively defined function symbol, or the application of a primitive operation.

Definition ($\mathbb{E}_{\mathtt{rdx}}$): The set of redexes, $\mathbb{E}_{\mathtt{rdx}}$, is defined as

$$\mathbb{E}_{\mathtt{rdx}} = \mathtt{if}(\mathbb{V},\mathbb{E},\mathbb{E}) \cup \mathtt{let}\{\mathbb{X} := \mathbb{V}\}\mathbb{E} \cup (\bigcup_{n\in\mathbb{N}} \mathbf{F}_n(\mathbb{V}^n) - \mathbb{P})$$

Note that the structured data, $\mathbb{P}$, are taken to be values (i.e are not redexes). Reduction contexts identify the subexpression of an expression that is to be evaluated next, they correspond to the standard reduction strategy (left-first, call-by-value) of [26] and were first introduced in [7].

Definition ($\mathbb{R}$): The set of reduction contexts, $\mathbb{R}$, is the subset of $\mathbb{C}$ defined by

$$\mathbb{R} = \{\bullet\} \cup \mathtt{let}\{\mathbb{X} := \mathbb{R}\}\mathbb{E} \cup \mathtt{if}(\mathbb{R},\mathbb{E},\mathbb{E}) \cup \bigcup_{n,m\in\mathbb{N}} \mathbf{F}_{n+m+1}(\mathbb{V}^n,\mathbb{R},\mathbb{E}^m)$$

We let R range over $\mathbb{R}$. An expression is either a value expression or decomposes uniquely into a primitive expression placed in a reduction context.

Definition ($\mathbb{M}$): The set of memory contexts, $\mathbb{M}$, is the set of contexts Γ of the form

$$\begin{array}{l} \mathtt{let}\{z_1 := \mathtt{mk}(\mathtt{Nil})\} \\ \quad \ddots \qquad\qquad \ddots \\ \qquad \mathtt{let}\{z_n := \mathtt{mk}(\mathtt{Nil})\}\mathtt{seq}(\mathtt{set}(z_1,v_1),\ldots,\mathtt{set}(z_n,v_n),\bullet) \end{array}$$

where $z_i \neq z_j$ when $i \neq j$. We include the possibility that $n = 0$, in which case $\Gamma = \bullet$. We let Γ range over $\mathbb{M}$.

We have divided the memory context into allocation, followed by assignment to allow for the construction of cycles. Thus, any state of memory is constructible by such an expression. We can view memory contexts as finite maps from variables to value expressions. Hence we define the domain of Γ (as above) to be $\mathrm{Dom}(\Gamma) = \{z_1,\ldots,z_n\}$, and $\Gamma(z_i) = v_i$ for $1 \leq i \leq n$. Two memory contexts are considered the same if they are the same when viewed as functions. Viewing memory contexts as finite maps, we define the modification of memory contexts, $\Gamma\{z := \mathtt{mk}(v)\}$, and the union of two memory contexts, $(\Gamma_0 \cup \Gamma_1)$, in the obvious way.

Definition ($\mathbb{D}$): The set of computation descriptions (briefly descriptions), $\mathbb{D}$, is defined to be the set $\mathbb{M} \times \mathbb{E}$. Thus a description is a pair with first component a memory context and second component an arbitrary expression. We let $\Gamma; e$ range over $\mathbb{D}$. A *closed* description is a description of the form $\Gamma; e$ where $\mathrm{Rng}(\Gamma) \cup \{e\} \subseteq \mathbb{E}_{\mathrm{Dom}(\Gamma)}$ (recall that $\mathbb{E}_{\mathrm{Dom}(\Gamma)}$ is the set of expressions whose free variables are among $\mathrm{Dom}(\Gamma)$). *Value descriptions* are descriptions whose expression component is a value expression, i.e. a description of the form $\Gamma; v$.

Single-step reduction ($\mapsto$) is a relation on *descriptions*. The reduction relation $\overset{*}{\mapsto}$ is the reflexive transitive closure of $\mapsto$. Officially, $\overset{*}{\mapsto}$ and $\mapsto$ should be parameterized by the definition system Δ, we will not make this parameter explicit here.

Definition ($\overset{*}{\mapsto}$): Single-step reduction is the least relation such that

(let) $\quad \Gamma; R[\mathtt{let}\{x := v\}e] \mapsto \Gamma; R[e^{\{x:=v\}}]$

(if) $\quad \Gamma; R[\mathtt{if}(v, e_1, e_2)] \mapsto \begin{cases} \Gamma; R[e_1] & \text{if } v \in (\mathbb{A} - \{\mathtt{Nil}\}) \cup \mathrm{Dom}(\Gamma) \\ \Gamma; R[e_2] & \text{if } v = \mathtt{Nil} \end{cases}$

(rec) $\quad \Gamma; R[f(v_1, \ldots, v_n)] \mapsto \Gamma; R[e^{\sigma}]$

$\qquad$ if $f(x_1, \ldots, x_n) \leftarrow e$ in Δ, $\mathrm{Dom}(\sigma) = \{x_1, \ldots, x_n\}$ and $\sigma(x_i) = v_i$

(atom) $\quad \Gamma; R[\mathtt{atom?}(v)] \mapsto \begin{cases} \Gamma; R[\mathtt{T}] & \text{if } v \in \mathbb{A} \\ \Gamma; R[\mathtt{Nil}] & \text{if } v \in \mathbb{P} \cup \mathrm{Dom}(\Gamma) \end{cases}$

(ispr) $\quad \Gamma; R[\mathtt{pr?}(v)] \mapsto \begin{cases} \Gamma; R[\mathtt{T}] & \text{if } v \in \mathbb{P} \\ \Gamma; R[\mathtt{Nil}] & \text{if } v \in \mathbb{A} \cup \mathrm{Dom}(\Gamma) \end{cases}$

(cell) $\quad \Gamma; R[\mathtt{cell?}(v)] \mapsto \begin{cases} \Gamma; R[\mathtt{T}] & \text{if } v \in \mathrm{Dom}(\Gamma) \\ \Gamma; R[\mathtt{Nil}] & \text{if } v \in \mathbb{P} \cup \mathbb{A} \end{cases}$

(eq) $\quad \Gamma; R[\mathtt{eq?}(v_0, v_1)] \mapsto \begin{cases} \Gamma; R[\mathtt{T}] & \text{if } v_0 = v_1,\ v_0, v_1 \in \mathrm{Dom}(\Gamma) \cup \mathbb{A}, \\ \Gamma; R[\mathtt{Nil}] & \text{otherwise and } \mathrm{FV}(v_0, v_1) \subseteq \mathrm{Dom}(\Gamma). \end{cases}$

(fst) $\quad \Gamma; R[\mathtt{fst}(\mathtt{pr}(v_1, v_2))] \mapsto \Gamma; R[v_1]$

(snd) $\quad \Gamma; R[\mathtt{snd}(\mathtt{pr}(v_1, v_2))] \mapsto \Gamma; R[v_2]$

(mk) $\quad \Gamma; R[\mathtt{mk}(v)] \mapsto \Gamma\{z := \mathtt{mk}(v)\}; R[z] \qquad$ if $z \notin \mathrm{Dom}(\Gamma) \cup \mathrm{FV}(R[v])$

(get) $\quad \Gamma; R[\mathtt{get}(z)] \mapsto \Gamma; R[v] \qquad$ if $z \in \mathrm{Dom}(\Gamma)$ and $\Gamma(z) = v$

(set) $\quad \Gamma; R[\mathtt{set}(z, v)] \mapsto \Gamma\{z := \mathtt{mk}(v)\}; R[\mathtt{Nil}] \qquad$ if $z \in \mathrm{Dom}(\Gamma)$

In the definition of $\mapsto$ we have not restricted our attention to closed descriptions, thus allowing symbolic or parametric computation. Note however that in the `atom?` and `cell?` rules if one of the arguments is a variable not in the domain of the memory context then the primitive reduction step is not determined (i.e. it suspends due to a lack of information). This is also the case in the `eq?`, `get`, and `set` rules.

Definition ($\downarrow \uparrow \updownarrow$): A closed description, $\Gamma; e$ is *defined* (written $\downarrow \Gamma; e$) if it evaluates to a value description. A description is undefined (written $\uparrow \Gamma; e$) if it

is not defined.

$$\downarrow(\Gamma; e) \Leftrightarrow (\exists \Gamma'; v')(\Gamma; e \overset{*}{\mapsto} \Gamma'; v')$$

$$\uparrow(\Gamma; e) \Leftrightarrow \neg\downarrow(\Gamma; e)$$

For closed expressions e, we write $\downarrow e$ to mean $\downarrow\emptyset; e$ and $e_0 \updownarrow e_1$ to mean that $\downarrow e_0$ iff $\downarrow e_1$.

2.3 Operational Equivalence of Terms

Operational equivalence formalizes the notion of equivalence as black-boxes. Treating programs as black boxes requires only observing what effects and values they produce, and not how they produce them. Our definition extends the extensional equivalence relations defined by [24] and [26] to computation over memory structures.

Definition ($\sqsubseteq \cong$): Two expressions are operationally approximate, written $e_0 \sqsubseteq e_1$, if for any closing context C, if $C[e_0]$ is defined then $C[e_1]$ is defined. Two expressions are operationally equivalent, written $e_0 \cong e_1$, if they approximate one another.

$$e_0 \sqsubseteq e_1 \Leftrightarrow (\forall C \in \mathbb{C} \mid C[e_0], C[e_1] \in \mathbb{E}_\emptyset)(\downarrow C[e_0] \Rightarrow \downarrow C[e_1])$$

$$e_0 \cong e_1 \Leftrightarrow e_0 \sqsubseteq e_1 \wedge e_1 \sqsubseteq e_0$$

The operational equivalence is not trivial since the inclusion of branching implies that `T` and `Nil` are not equivalent. By definition operational equivalence is a congruence relation on expressions:

Lemma (Congruence):

$$e_0 \cong e_1 \Leftrightarrow (\forall C \in \mathbb{C})(C[e_0] \cong C[e_1])$$

In general it is very difficult to establish the operational equivalence of expressions. Thus it is desirable to have a simpler characterization of $\cong$. There are two simple characterizations of operational equivalence available in this setting. The first is called (**ciu**) equivalence and also holds for the full $\lambda_{\mathtt{mk}}$ calculus. The second is called strong isomorphism and holds only in the first order case.

The first result is that two expressions are operationally equivalent just if all closed instantiations of all uses are equidefined. This latter property is a weak form of extensionality.

Theorem (ciu [18, 10]):

$$e_0 \cong e_1 \Leftrightarrow (\forall \Gamma, \sigma, R \mid \mathrm{FV}(\Gamma[R[e_i^\sigma]]) = \emptyset)(\Gamma[R[e_0^\sigma]] \updownarrow \Gamma[R[e_1^\sigma]])$$

For the second we say two expressions e_0 and e_1 are strongly isomorphic if for every closed instantiation either both are undefined or both are defined and evaluate to objects that are equal modulo the production of garbage. By garbage

we mean cells constructed in the process of evaluation that are not accessible from either the result or the domain of the initial memory. The result then is that operational equivalence and strong isomorphism coincide.

Theorem (striso [13, 14]): $e_0 \cong e_1$ iff if for each Γ, σ such that $\Gamma[e_j^\sigma] \in \mathbb{E}_\emptyset$ for $j < 2$, one of the following holds:

(1) $\uparrow(\Gamma; e_0^\sigma)$ and $\uparrow(\Gamma; e_1^\sigma)$, or

(2) there exist v, Γ', Γ_0, Γ_1 such that $\mathrm{Dom}(\Gamma) \subseteq \mathrm{Dom}(\Gamma')$, $\Gamma'[v] \in \mathbb{E}_\emptyset$, $\mathrm{Dom}(\Gamma') \cap \mathrm{Dom}(\Gamma_j) = \emptyset$ and $\Gamma; e_j^\sigma \overset{*}{\mapsto} (\Gamma_j \cup \Gamma'); v$ for $j < 2$.

Using these theorems we can easily establish, for example, the validity of the let-rules of the lambda-c calculus [23]

Corollary ($\mathtt{let}_c$):

(i) $e^{\{x:=v\}} \cong \mathtt{let}\{x := v\}e$

(ii) $R[e] \cong \mathtt{let}\{x := e\}R[x]$

(iii) $R[\mathtt{let}\{x := e_0\}e_1] \cong \mathtt{let}\{x := e_0\}R[e_1]$

where in (ii) and (iii) we require x not free in R.

Another nice property that is easily established is that reduction preserves operational equivalence.

Lemma (eval): $\Gamma; e \mapsto \Gamma'; e' \;\Rightarrow\; \Gamma[e] \cong \Gamma'[e']$.

3 The Syntax and Semantics of the First-Order Theory

3.1 Syntax of Formulas

The atomic formulas of our language assert the operational equivalence of two expressions. In addition to the usual first-order formula constructions we add *contextual assertions:* if Φ is a formula and U is a certain type of context, then $U[\![\Phi]\!]$ is a formula. This form of formula expresses the fact that the assertion Φ holds at the point in the program text marked by the hole in U, if execution of the program reaches that point. The contexts allowed in contextual assertions are called *univalent contexts*, (U-contexts). The class of U-contexts, $\mathbb{U}$, is defined as follows.

Definition ($\mathbb{U}$):

$$\mathbb{U} = \{\bullet\} + \mathtt{let}\{\mathbb{X} := \mathbb{E}\}\mathbb{U}$$

The well-formed formulas, $\mathbb{W}$, of (the first order part of) our logic are defined as follows:

Definition ($\mathbb{W}$):

$$\mathbb{W} = (\mathbb{E} \cong \mathbb{E}) + (\mathbb{W} \;\Rightarrow\; \mathbb{W}) + (\mathbb{U}[\![\mathbb{W}]\!]) + (\forall \mathbb{X})(\mathbb{W})$$

We let Φ range over $\mathbb{W}$. Negation is definable, $\neg\Phi$ is just $\Phi \Rightarrow \mathtt{False}$, where $\mathtt{False}$ is any unsatisfiable assertion, such as $\mathtt{T} \cong \mathtt{Nil}$. Similarly conjunction, $\wedge$, and disjunction, $\vee$ and the biconditional, $\Leftrightarrow$, are all definable in the usual manner. We let $\Downarrow e$ abbreviate $\neg(\mathtt{seq}(e, \bullet)[\![\mathtt{False}]\!])$ and $\Uparrow e$ abbreviate its negation. Note that $\Downarrow e$ expresses the computational definedness of the expression e.

Given a particular U, for example $\mathtt{let}\{x := \mathtt{mk}(v)\}\bullet$, we will often abuse notation and write $\mathtt{let}\{x := \mathtt{mk}(v)\}[\![\Phi]\!]$ rather than $(\mathtt{let}\{x := \mathtt{mk}(v)\}\bullet)[\![\Phi]\!]$. Thus we write $\neg\mathtt{seq}(e, [\![\mathtt{False}]\!])$ rather than $\neg(\mathtt{seq}(e, \bullet)[\![\mathtt{False}]\!])$.

Note that the context U will in general bind free variables in Φ. A simple example is the law which expresses the effects of $\mathtt{mk}$:

$$(\forall y)(\mathtt{let}\{x := \mathtt{mk}(v)\}[\![\neg(x \cong y) \wedge \mathtt{cell?}(x) \cong \mathtt{T} \wedge \mathtt{get}(x) \cong v]\!])$$

For simplicity we have omitted certain possible contexts from the definition of $\mathbb{U}$. However those left out may be considered abbreviations. Two examples are:

(1) $\mathtt{if}(e_0, [\![\Phi_0]\!], [\![\Phi_1]\!])$ abbreviates

$$\mathtt{let}\{z := e_0\}[\![(z \cong \mathtt{Nil} \Rightarrow \Phi_1) \wedge (\neg(z \cong \mathtt{Nil}) \Rightarrow \Phi_0)]\!] \qquad z \text{ fresh.}$$

(2) $\vartheta(e_0, \ldots, e_n, U[\![\Phi]\!], e_{n+1}, \ldots)$ abbreviates $\mathtt{seq}(e_0, \ldots, e_n, U[\![\Phi]\!])$

In order to define the semantics of contextual assertions, we must extend computation to univalent contexts. The idea here is quite simple, to compute with contexts we need to keep track of the $\mathtt{let}$-conversions that have taken place with the hole in the scope of the $\mathtt{let}$. To indicate that the substitution σ is in force at the hole in U we write $U[\![\sigma]\!]$. Computation is then written as $\Gamma; U[\![\sigma]\!] \overset{*}{\mapsto} \Gamma'; U'[\![\sigma']\!]$ and is defined in full in [10]. For example if x is not in the domain of σ, then

$$\Gamma; \mathtt{let}\{x := v\}[\![\sigma]\!] \overset{*}{\mapsto} \Gamma; [\![\sigma\{x := v\}]\!].$$

3.2 Semantics of Formulas

In addition to being a useful tool for establishing laws of operational equivalence, (**ciu**) can be used to define a satisfaction relation between memory contexts and equivalence assertions. In an obvious analogy with the usual first-order Tarskian definition of satisfaction this can be extended to define a satisfaction relation $\Gamma \models \Phi[\sigma]$.

The definition of satisfaction $\Gamma \models \Phi[\sigma]$ is given by a simple induction on the structure of Φ.

Definition ($\Gamma \models \Phi[\sigma]$): $(\forall \Gamma, \sigma, \Phi, e_j)$ such that $\mathrm{FV}(\Phi^\sigma) \cup \mathrm{FV}(e_j^\sigma) \subseteq \mathrm{Dom}(\Gamma)$ for $j < 2$ we define satisfaction:

$\Gamma \models (e_0 \cong e_1)[\sigma]$ iff $(\forall R \in \mathbb{R}_{\mathrm{Dom}(\Gamma)})(\Gamma[R[e_0^\sigma]] \updownarrow \Gamma[R[e_1^\sigma]])$

$\Gamma \models (\Phi_0 \Rightarrow \Phi_1)[\sigma]$ iff $(\Gamma \models \Phi_0[\sigma])$ implies $(\Gamma \models \Phi_1[\sigma])$

$\Gamma \models U[\![\Phi]\!][\sigma]$ iff $(\forall \Gamma', R, \sigma')((\Gamma; U[\![\sigma]\!] \overset{*}{\mapsto} \Gamma'; R[\![\sigma']\!])$ implies $\Gamma' \models \Phi[\sigma'])$

$\Gamma \models (\forall x)\Phi[\sigma]$ iff $(\forall v \in \mathbb{V}_{\mathrm{Dom}(\Gamma)})(\Gamma \models \Phi[\sigma\{x := v\}])$

We say that a formula is *valid*, written $\models \Phi$, if $\Gamma \models \Phi[\sigma]$ for Γ, σ such that $\mathrm{FV}(\Phi^\sigma) \subseteq \mathrm{Dom}(\Gamma)$. Following the usual convention we will often write Φ as an assertion that Φ is valid, omitting the $\models$ sign. Note that the underlying logic is completely classical.

3.3 Contextual Assertion Principles

The theorem (**ca**) provides several principles for reasoning about contextual assertions: a simple principle concerning reduction contexts; a general principle for introducing contextual assertions (akin to the rule of necessitation in modal logic); a principle for propagating contextual assertions through equations; and a principle for composing contexts (or collapsing nested contextual assertions); a principle for manipulating contexts; three principles demonstrating that contextual assertions interact nicely with the propositional connectives, if we take proper account of assertions that are true for the trivial reason that during execution, the point in the program text marked by the context hole is never reached; and a principle (whose converse is false) concerning the quantifier. Finally a lemma demonstrating that contextual assertions interact nicely with evaluation. Proofs of the principles in this section can be found in [10].

Theorem (ca):

(i) $\quad \Phi \Rightarrow R[\![\Phi]\!]$

(ii) $\quad \models \Phi \quad \text{implies} \quad \models U[\![\Phi]\!]$

(iii) $\quad U[\![e_0 \cong e_1]\!] \Rightarrow U[e_0] \cong U[e_1]$

(iiii) $\quad \models U_0[\![U_1[\![\Phi]\!]]\!] \Leftrightarrow (U_0[U_1])[\![\Phi]\!]$

(iv) $\quad \mathtt{let}\{x := R[e]\}U[\![\Phi]\!] \Leftrightarrow \mathtt{let}\{y := e\}\mathtt{let}\{x := R[y]\}U[\![\Phi]\!] \qquad y$ fresh

(v) $\quad U[\![\mathtt{False}]\!] \Rightarrow U[\![\Phi]\!]$

(vii) $\quad U[\![\neg\Phi]\!] \Leftrightarrow (U[\![\mathtt{False}]\!] \vee \neg U[\![\Phi]\!])$

(viii) $\quad U[\![\Phi_0 \Rightarrow \Phi_1]\!] \Leftrightarrow (U[\![\Phi_0]\!] \Rightarrow U[\![\Phi_1]\!])$

(viiii) $\quad U[\![\forall x \Phi]\!] \Rightarrow \forall x\, U[\![\Phi]\!] \qquad$ where x not free in U

Lemma (eval):

$$\Gamma_0; U_0[\![\sigma_0]\!] \overset{*}{\mapsto} \Gamma_1; U_1[\![\sigma_1]\!] \quad \text{implies} \quad (\Gamma_0 \models U_0[\![\Phi]\!][\sigma_0] \quad \text{iff} \quad \Gamma_1 \models U_1[\![\Phi]\!][\sigma_1])$$

A simple use of (**ca**) is the following principle:

Lemma (cut):

$$\models \Phi \Rightarrow U[\![\Phi']\!] \quad \text{and} \quad \models \Phi' \Rightarrow U'[\![\Phi'']\!] \quad \text{implies} \quad \models \Phi \Rightarrow (U[U'])[\![\Phi'']\!]$$

Proof (cut): Assume $\models \Phi' \Rightarrow U'[\![\Phi'']\!]$. Thus

$\models U[\![\Phi' \Rightarrow U'[\![\Phi'']\!]]\!]$ by (**ca.ii**).

$\models U[\![\Phi']\!] \Rightarrow U[\![U'[\![\Phi'']\!]]\!]$ by (**ca.viii**).

$\models \Phi \Rightarrow (U[U'])[\![\Phi'']\!]$ by (**ca.iiii**) and classical logic.

3.4 Memory Operation Principles

This logic extends and improves the complete first order system presented in [16, 19]. There certain reasoning principles were established as basic, and from these all others, suitably restricted, could be derived using simple equational reasoning. The system presented there had several defects. In particular the rules concerning the effects of `mk` and **set** had complicated side-conditions. Using contextual assertions we can express them simply and elegantly. Their justification is also unproblematic.

The contextual assertions and laws involving `mk`, **set** and **get** are given below. The assertion, (`mk`.**i**), describes the allocation effect of a call to `mk`. While (`mk`.**ii**) expresses what is unaffected by a call to `mk`. The assertion, (`mk`.**iii**), expresses the totality of `mk`. The `mk` delay law, (`mk`.**iv**), asserts that the time of allocation has no discernable effect on the resulting cell. In a world with control effects evaluation of e_0 must be free of them for this principle to be valid [6]. The first three contextual assertions regarding **set** are analogous to those of `mk`. They describe what is returned and what is altered, what is not altered as well as when the operation is defined. The remaining three principles involve the commuting, cancellation, absorption of calls to **set**. For example the **set** absorption principle, (**set.vi**), expresses that under certain simple conditions allocation followed by assignment may be replaced by a suitably altered allocation. The contextual assertions regarding **get** follow the above pattern. They describe what is altered and returned, what is not altered as well as when the operation is defined.

Lemma (Memory Operation Laws):

(`mk`.i) $\mathtt{let}\{x := \mathtt{mk}(v)\}[\![\neg(x \cong y) \wedge \mathtt{cell?}(x) \cong \mathtt{T} \wedge \mathtt{get}(x) \cong v]\!]$

x fresh

(`mk`.ii) $y \cong \mathtt{get}(z) \Rightarrow \mathtt{let}\{x := \mathtt{mk}(v)\}[\![y \cong \mathtt{get}(z)]\!]$

(`mk`.iii) $\Downarrow \mathtt{mk}(z)$

(`mk`.iv) $\mathtt{let}\{y := e_0\}\mathtt{let}\{x := \mathtt{mk}(v)\}e_1 \cong \mathtt{let}\{x := \mathtt{mk}(v)\}\mathtt{let}\{y := e_0\}e_1$

$x \notin \mathrm{FV}(e_0)$, $y \notin \mathrm{FV}(v)$

(**set**.i) $\mathtt{cell?}(z) \Rightarrow \mathtt{let}\{x := \mathtt{set}(z, y)\}[\![\mathtt{get}(z) \cong y \wedge x \cong \mathtt{Nil}]\!])$

(**set**.ii) $(y \cong \mathtt{get}(z) \wedge \neg(w \cong z)) \Rightarrow \mathtt{let}\{x := \mathtt{set}(w, v)\}[\![y \cong \mathtt{get}(z)]\!]$

(**set**.iii) $\mathtt{cell?}(z) \Rightarrow \Downarrow \mathtt{set}(z, x)$

(set.iv) $\neg(x_0 \cong x_2) \Rightarrow$
$\mathtt{seq}(\mathtt{set}(x_0, x_1), \mathtt{set}(x_2, x_3)) \cong \mathtt{seq}(\mathtt{set}(x_2, x_3), \mathtt{set}(x_0, x_1))$

(set.v) $\mathtt{seq}(\mathtt{set}(x, y_0), \mathtt{set}(x, y_1)) \cong \mathtt{set}(x, y_1)$

(set.vi) $\mathtt{let}\{z := \mathtt{mk}(x)\}\mathtt{seq}(\mathtt{set}(z, w), e) \cong \mathtt{let}\{z := \mathtt{mk}(w)\}e$

z not free in w

(get.i) $\mathtt{let}\{x := \mathtt{get}(y)\}[\![x \cong \mathtt{get}(y)]\!]$

(get.ii) $y \cong \mathtt{get}(z) \Rightarrow \mathtt{let}\{x := \mathtt{get}(w)\}[\![y \cong \mathtt{get}(z)]\!]$

(get.iii) $\mathtt{cell?}(x) \Leftrightarrow (\exists y)(\mathtt{get}(x) \cong y)$

We should also point out that officially we should make Δ a parameter of the satisfaction relation but, as in the presentation of the operational semantics, we will not usually make this parameter explicit. The only rule which depends on the definition system is the following unfolding rule, (**U**), it corresponds to the (**rec**) rule for single-step reduction.

Lemma (Unfolding law):

$$f(e_1, \ldots, e_n) \cong \mathtt{let}\{x_1 := e_1\} \ldots \mathtt{let}\{x_n := e_n\}e$$

where $f(x_1, \ldots x_n) \leftarrow e$ is in Δ and x_i are chosen fresh.

A simple example of a proof is the following lemma.

Lemma (if delay): If $z \notin \mathrm{FV}(e)$, then

$$\mathtt{let}\{z := \mathtt{mk}(x)\}\mathtt{if}(e, e_1, e_2) \cong \mathtt{if}(e, \mathtt{let}\{z := \mathtt{mk}(x)\}e_1, \mathtt{let}\{z := \mathtt{mk}(x)\}e_2)$$

Proof (if delay):

$\mathtt{let}\{z := \mathtt{mk}(x)\}\mathtt{if}(e, e_1, e_2)$

$\cong \mathtt{let}\{z := \mathtt{mk}(x)\}\mathtt{let}\{y := e\}\mathtt{if}(y, e_1, e_2)$

by (**congruence**) and ($\mathtt{let}_c$**.ii**)

$\cong \mathtt{let}\{y := e\}\mathtt{let}\{z := \mathtt{mk}(x)\}\mathtt{if}(y, e_1, e_2)$ by (**mk.iv**)

So it suffices to show that if $z \notin \mathrm{FV}(e)$, then

$$\mathtt{let}\{z := \mathtt{mk}(x)\}\mathtt{if}(y, e_1, e_2) \cong \mathtt{if}(y, \mathtt{let}\{z := \mathtt{mk}(x)\}e_1, \mathtt{let}\{z := \mathtt{mk}(x)\}e_2)$$

First observe that static properties always propagate:

(i) $\neg(y \cong \mathtt{Nil}) \Rightarrow \mathtt{let}\{z := \mathtt{mk}(x)\}[\![\neg(y \cong \mathtt{Nil})]\!]$

(ii) $\neg(y \cong \mathtt{Nil}) \Rightarrow \mathtt{if}(y, e_1, e_2) \cong e_1$

Putting (**i**) and (**ii**) together via (**cut**) gives

$$\neg(y \cong \mathtt{Nil}) \Rightarrow \mathtt{let}\{z := \mathtt{mk}(x)\}\mathtt{if}(y, e_1, e_2) \cong \mathtt{let}\{z := \mathtt{mk}(x)\}e_1$$

A second application of (**ii**) gives

$$\neg(y \cong \texttt{Nil}) \Rightarrow$$
$$\texttt{let}\{z := \texttt{mk}(x)\}\texttt{if}(y, e_1, e_2)$$
$$\cong$$
$$\texttt{if}(y, \texttt{let}\{z := \texttt{mk}(x)\}e_1, \texttt{let}\{z := \texttt{mk}(x)\}e_2)$$

Similar reasoning under the hypothesis that $y \cong \texttt{Nil}$ provides

$$y \cong \texttt{Nil} \Rightarrow$$
$$\texttt{let}\{z := \texttt{mk}(x)\}\texttt{if}(y, e_1, e_2)$$
$$\cong$$
$$\texttt{if}(y, \texttt{let}\{z := \texttt{mk}(x)\}e_1, \texttt{let}\{z := \texttt{mk}(x)\}e_2)$$

Classical logic provides the rest. □

3.5 Reasoning about Newness

A very useful concept is that of a cell being new. A cell is considered *new* in a memory if it appears as if it was newly allocated, i.e. it is does not appear elsewhere in memory. We will use the newness assertions to recycle cells. Since a *new* cell is as good as a newly allocated cell. The definition of new requires an auxiliary program, *notin*, that checks whether z is not accessible from x via a chain of projections.

$$\begin{aligned} \mathit{notin}(z, x) \leftarrow \texttt{cond}([&\texttt{not}(\texttt{cell?}(z)), \texttt{T}] \\ &[\texttt{cell?}(x), \texttt{not}(\texttt{eq?}(x, z))] \\ &[\texttt{atom?}(x), \texttt{T}] \\ &[\texttt{pr?}(x), \texttt{and}(\mathit{notin}(z, \texttt{fst}(x)), \mathit{notin}(z, \texttt{snd}(x)))]) \end{aligned}$$

For convenience will will often write x *notin* y to abbreviate the assertion $\mathit{notin}(x, y) \cong \texttt{T}$.

Definition (new):

$$\texttt{new}(x) \quad \text{abbreviates} \quad \texttt{cell?}(x) \cong \texttt{T} \wedge (\forall y)(\texttt{cell?}(y) \Rightarrow x \ \mathit{notin}\ \texttt{get}(y))$$

A simple example of *newness* is given by the following lemma. It states that a cell is new immediately after it is created.

Lemma (new creation):

$$\texttt{let}\{x := \texttt{mk}(v)\}[\![\texttt{new}(x)]\!]$$

The next lemma provides some principles for propagating newness.

Lemma (new):

(i) $\mathtt{new}(x) \Rightarrow \mathtt{let}\{z := \vartheta(\bar{v})\}[\![\mathtt{new}(x)]\!] \qquad \vartheta \notin \{\mathtt{mk}, \mathtt{set}\}$

(ii) $\mathtt{new}(x) \wedge (x \ \mathit{notin}\ v) \Rightarrow \mathtt{let}\{z := \mathtt{mk}(v)\}[\![\mathtt{new}(x)]\!]$

(iii) $\mathtt{new}(x) \wedge (x \ \mathit{notin}\ v) \Rightarrow \mathtt{let}\{z := \mathtt{set}(y, v)\}[\![\mathtt{new}(x)]\!]$

(iiii) $U[\![\mathtt{new}(x)]\!] \Rightarrow (R[U])[\![\mathtt{new}(x)]\!]$

(iv) $\mathtt{seq}(e, [\![\mathtt{new}(x)]\!]) \Rightarrow \mathtt{if}(e, [\![\mathtt{new}(x)]\!], [\![\mathtt{new}(x)]\!])$

(v) $\mathtt{new}(x) \wedge \bigwedge_{w \in \mathbf{FV}(e)} x \ \mathit{notin}\ w \Rightarrow \mathtt{let}\{y := e\}[\![\mathtt{new}(x)]\!]$

Note that in (**new.iii**) the variables x and y could name the same cell. An important principle concerning newness is *reuse*:

Theorem (new reuse):

$$\begin{aligned}
&\mathtt{let}\{x := \mathtt{mk}(v)\}\mathtt{seq}(e_0, [\![\mathtt{new}(x)]\!]) \Rightarrow \\
&\quad (\mathtt{let}\{x := \mathtt{mk}(v)\}\mathtt{seq}(e_0, e_1) \\
&\qquad \cong \\
&\quad \mathtt{seq}(\mathtt{let}\{x := \mathtt{mk}(v)\}e_0, \mathtt{let}\{x := \mathtt{mk}(\mathtt{get}(x))\}e_1))
\end{aligned}$$

4 Tools for Transformations

We now develop some tools for reasoning about recursively defined programs. These tools fall into three general classes: principles for proving input-output assertions, principles for proving program equivalence, and principles for manipulating memory. Contextual assertions play an important role here, especially in the formulation of rules for propagation of invariants.

We begin with some notation. In the following we let $\mathbf{x}$, $\mathbf{x}_i$, $\mathbf{y}$ be sequences of distinct variables, and let $\mathbf{v}$ be a sequence of value expressions. We let $\mathbf{f}$ be a sequence of function symbols and let f_i denote the i-th symbol of $\mathbf{f}$. We work in the context of an arbitrary but fixed system of definitions Δ. If the symbols of $\mathbf{f}$ are among those defined in Δ, we write $\Delta_{\mathbf{f}} \subseteq \Delta$ for the subsystem $[f_i(\mathbf{x}_i) \leftarrow F_i \mid i \leq n]$ where $n+1$ is the length of $\mathbf{f}$, and $f_i(\mathbf{x}_i) \leftarrow F_i$ is the definition of f_i in Δ for $i \leq n$.

We write $e\{f := \lambda\mathbf{x}.e'\}$ for the replacement of the function symbol f by the explicitly defined function $\lambda\mathbf{x}.e'$ in e. This is obtained by replacing each occurrence $f(e_1, \ldots, e_k)$ in e by

$$\mathtt{let}\{x_i := e_i \mid 1 \leq i \leq k\}e'.$$

Similarly, $e\{f_i := \lambda\mathbf{x}_i.e_i'\}$ denotes the simultaneous replacement of function symbols f_i by $\lambda\mathbf{x}_i.e_i'$ in e. We will use analogous conventions for sequences, $\mathbf{g}$ of function symbols.

4.1 Invariants

The central idea in this subsection is to formalize the notion of propagating an assertion to all important points in an expression, which in this case consists in all occurrences of recursive function calls. This is accomplished by defining the notion of a system of invariants, generalizing Floyd-style inductive assertions.

Definition (System of invariants): A system of invariants for $\mathbf{f}$ is a sequence of pairs of formulas (ϕ_i, ψ_i) for $i \leq n = |\,\mathbf{f}\,|$.

We let I range over systems of invariants. The ϕ_i are to be thought of as pre-conditions for f_i, while the ψ_i are post-conditions. We adopt the convention that the variable z will serve as the value of the output, and that $\mathbf{x}_i$ will serve as input variables for ϕ_i. Thus occurrences of these variables in the invariants will be used to name parameters in calls to the defined functions.

These invariants are used to propagate assertions across any calls to these functions. To make this precise we make the following definition.

Definition (I, $\mathbf{f}$-propagation): Let I be a system of invariants for $\mathbf{f}$, and let θ, θ' be any invariants. We say that θ $(I, \mathbf{f})$-propagates across $\mathtt{let}\{z := e\}\bullet$ to θ' and denote this relation by

$$\models_{I,\mathbf{f}} \theta \Rightarrow \mathtt{let}\{z := e\}[\![\theta']\!]$$

just if one of the following holds:

(i) No element of $\mathbf{f}$ occurs in e and $\theta \Rightarrow \mathtt{let}\{z := e\}[\![\theta']\!]$.

(ii) $e = f_i(\mathbf{v})$, $\theta \Rightarrow (\phi_i)^\sigma$ and $(\psi_i)^\sigma \Rightarrow \theta'$, where $\sigma = \{\mathbf{x}_i := \mathbf{v}\}$;

(iii) $e = \mathtt{let}\{z_0 := e_0\}e_1$ and there is some θ_0 such that

$$\models_{I,\mathbf{f}} \theta \Rightarrow \mathtt{let}\{z := e_0\}[\![\theta_0^{\{z_0:=z\}}]\!]$$

$$\models_{I,\mathbf{f}} \theta_0 \Rightarrow \mathtt{let}\{z := e_1\}[\![\theta']\!]$$

(iv) $e = \mathtt{if}(e_0, e_1, e_2)$ and there is some θ_1, θ_2 such that

$$\theta \Rightarrow \mathtt{if}(e_0, [\![\theta_1]\!], [\![\theta_2]\!])$$

$$\models_{I,\mathbf{f}} \theta_1 \Rightarrow \mathtt{let}\{z := e_1\}[\![\theta']\!]$$

$$\models_{I,\mathbf{f}} \theta_2 \Rightarrow \mathtt{let}\{z := e_2\}[\![\theta']\!]$$

The key computational property of invariant propagation is expressed by the following lemma.

Lemma (I-prop): Let I be a system of invariants for $\mathbf{f}$, and let θ, θ' be invariants such that $\models_{I,\mathbf{f}} \theta \Rightarrow \mathtt{let}\{z := e\}[\![\theta']\!]$. If $\Gamma \models \theta[\sigma]$, and $\Gamma; e^\sigma \overset{*}{\mapsto} \Gamma'; v$ then

(a) $\Gamma; e^\sigma \overset{*}{\mapsto} \Gamma'; v$ without using $\Delta_{\mathbf{f}}$ and $\Gamma' \models \theta'[\sigma\{z := v\}]$; or

(b) $\Gamma; e^\sigma \overset{*}{\mapsto} \Gamma'; R[f_i(\mathbf{v})]$ (without using $\Delta_{\mathbf{f}}$) and

(b.1) $\Gamma' \models \phi_i[\{\mathbf{x} := \mathbf{v}\}]$;

(b.2) $\models_{I,\mathbf{f}} \psi_i\{\mathbf{x} := \mathbf{v}\} \Rightarrow \mathtt{let}\{z := R[z]\}[\![\theta']\!]$

Definition (I-inductive): Let I be a system of invariants for $\mathbf{f}$. We say that $\Delta_{\mathbf{f}}$ is I-inductive if $\models_{I,\mathbf{f}} \phi_i \Rightarrow \mathtt{let}\{z := F_i\}[\![\psi_i]\!]$ for $i \leq n$ (recall that $\Delta_{\mathbf{f}} = [f_i(\mathbf{x}_i) \leftarrow F_i \mid i \leq n]$).

Theorem (Subgoal Induction): Let I be a system of invariants for $\mathbf{f}$. To prove

$$\phi_i \Rightarrow \mathtt{let}\{z := f_i(\mathbf{x}_i)\}[\![\psi_i]\!]$$

for $i \leq n$, it suffices to prove that $\Delta_{\mathbf{f}}$ is I-inductive.

Proof : The validity of this principle is established by a simple computation induction, using (**I-prop**), showing the more general fact that if $\Delta_{\mathbf{f}}$ is I-inductive then $\models_{I,\mathbf{f}} \theta \Rightarrow \mathtt{let}\{z := e\}[\![\theta']\!]$ implies $\theta \Rightarrow \mathtt{let}\{z := e\}[\![\theta']\!]$. □

Corollary (Subgoal Induction): To prove $\theta \Rightarrow \mathtt{let}\{z := f_0(\mathbf{x}_0)\}[\![\theta']\!]$ it suffices to find a system of invariants I for $\mathbf{f}$, such that $\theta \Rightarrow \phi_0$, $\psi_0 \Rightarrow \theta'$, and $\Delta_{\mathbf{f}}$ is I-inductive.

As a simple example of the use of subgoal induction, we define $\mathtt{sum!}(n, c)$ that adds the numbers from 1 to n to the contents of c. We show that sum! returns c (call-by-reference), and preserves newness of c, and c always contains an number.

Definition (sum!):

$$\mathtt{sum!}(n, c) \leftarrow \mathtt{if}(\mathtt{eq?}(n, 0), c, \mathtt{seq}(\mathtt{set}(c, n + \mathtt{get}(c)), \mathtt{sum!}(n - 1, c)))$$

Lemma (sum!):

$$(\mathtt{new}(c) \wedge \mathtt{nat?}(\mathtt{get}(c)) \cong \mathtt{T}) \Rightarrow$$
$$\mathtt{let}\{z := \mathtt{sum!}(n, c)\}[\![\mathtt{new}(c) \wedge \mathtt{nat?}(\mathtt{get}(c)) \cong \mathtt{T} \wedge c \cong z]\!]$$

Proof : This is proved by subgoal induction. Let

$\theta = \phi_0 = \mathtt{new}(c) \wedge \mathtt{nat?}(\mathtt{get}(c)) \cong \mathtt{T}$,

$\psi_0 = \theta' = \mathtt{new}(c) \wedge \mathtt{nat?}(\mathtt{get}(c)) \cong \mathtt{T} \wedge c \cong z$,

be the system of invariants for sum!. We must show that $\Delta_{\mathtt{sum!}}$ is I-inductive, i.e.

$$\models_{I,\mathtt{sum!}} \phi_0 \Rightarrow \mathtt{let}\{z := F_{\mathtt{sum!}}\}[\![\psi_0]\!]$$
where $F_{\mathtt{sum!}} = \mathtt{if}(\mathtt{eq?}(n, 0), c, \mathtt{seq}(\mathtt{set}(c, n + \mathtt{get}(c)), \mathtt{sum!}(n - 1, c)))$.

By clause (iv) of the definition $\models_{I,\mathtt{sum!}}$ this reduces to showing

(0) $\models_{I,\mathtt{sum!}} \theta_0 \Rightarrow \mathtt{let}\{x := \mathtt{eq?}(n, 0)\}[\![\neg(x \cong \mathtt{Nil}) \Rightarrow \theta_1 \wedge x \cong \mathtt{Nil} \Rightarrow \theta_2]\!]$

(1) $\models_{I,\mathtt{sum!}} \theta_1 \Rightarrow \mathtt{let}\{z := c\}[\![\psi_0]\!]$

(2) $\models_{I,\mathtt{sum!}} \theta_2 \Rightarrow \mathtt{let}\{z := \mathtt{seq}(\mathtt{set}(c, n + \mathtt{get}(c)), \mathtt{sum!}(n - 1, c))\}[\![\psi_0]\!]$

for some θ_1, θ_2. We take $\theta_1 = \theta_0 \wedge \mathtt{eq?}(n, 0) \cong \mathtt{T}$ and $\theta_2 = \theta_0 \wedge \mathtt{eq?}(n, 0) \cong \mathtt{Nil}$. Then (0) follows by clause (i), (**new.i**), (**ca**), and simple let laws. (1) follows by clause (i), (**new.i**), and simple let laws. For (2) we use clause (iii). Thus we must show

(3) $\quad \models_{I, \mathtt{sum!}} \phi_0 \Rightarrow \mathtt{let}\{z' := \mathtt{set}(c, n + \mathtt{get}(c))\}[\![\theta_0]\!]$

(4) $\quad \models_{I, \mathtt{sum!}} \theta_0 \Rightarrow \mathtt{let}\{z := \mathtt{sum!}(n-1, c))\}[\![\psi_0]\!]$

for some θ_0. Take $\theta_0 = \phi_0$, then (3) follows by clause (i), some simple properties of numbers and addition (c *notin* $n + \mathtt{get}(c)$) and (**new.iii**), and (4) follows by clause (ii).

□

4.2 General Principles for Establishing Equivalence

The recursion induction principle is based on the least-fixed-point semantics of recursive definition systems. Two systems can be shown to define equivalent programs by showing that each satisfies the others defining equations.

Theorem (Recursion Induction): To prove

$$\theta_i \Rightarrow f_i(\mathbf{x}_i) \cong g_i(\mathbf{x}_i)$$

for $i \leq n$, it suffices to find a system I of invariants for $\mathbf{f}$ (and $\mathbf{g}$), such that for $i \leq n$

(i) $\theta_i \Rightarrow \phi_i$;

(ii) $\Delta_{\mathbf{f}}$ and $\Delta_{\mathbf{g}}$ are I-inductive;

(iii) $\phi_i \Rightarrow g_i(\mathbf{x}_i) \cong F_i\{f_j := \lambda \mathbf{x}_j . g_j(\mathbf{x}_j) \mid j \leq n\}$, using $\Delta_{\mathbf{g}}$; and

(iv) $\phi_i \Rightarrow f_i(\mathbf{x}_i) \cong G_i\{g_j := \lambda \mathbf{x}_j . f_j(\mathbf{x}_j) \mid j \leq n\}$, using $\Delta_{\mathbf{f}}$.

Note that the condition (iii) says that $\mathbf{g}$ satisfies the defining equations for $\mathbf{f}$ and hence establishes that $f_i \sqsubseteq g_i$. Similarly (iv) establishes $g_i \sqsubseteq f_i$ (cf §2.3). As is the case of (**subgoal induction**) the validity of recursion induction is established by simple computation induction. This basic form of recursion induction can be generalized in many ways. The next theorem gives a generalization that allows for the functions being compared to have different argument lists.

Theorem (Recursion induction – elaborated): To prove

$$\theta_i \Rightarrow f_i(\mathbf{x}_i) \cong g_i(\mathbf{y}_i)$$

for $i \leq n$, (with the argument lists for for $\mathbf{g}$ possibly being different than those for $\mathbf{f}$) it suffices to find a system of invariants $I = [(\phi_i, \psi_i), i \leq n]$ and expressions $e_i^{\mathbf{f}}$, $e_i^{\mathbf{g}}$ such that for $i \leq n$

(i) $\theta_i \Rightarrow \phi_i$

(ii) $\Delta_{\mathbf{f}}$ and $\Delta_{\mathbf{g}}$ are I-inductive (taking the formal parameters to be $\mathbf{y}_i$ in the case of $\Delta_{\mathbf{g}}$).

(iii) $\phi_i \Rightarrow g_i(\mathbf{y}_i) \cong e_i^{\mathbf{g}} \cong F_i\{f_i := \lambda \mathbf{x}_i.e_i^{\mathbf{g}}, i \leq n\}$ using $\Delta_{\mathbf{g}}$,

(iv) $\phi_i \Rightarrow f_i(\mathbf{x}_i) \cong e_i^{\mathbf{f}} \cong G_i\{g_i := \lambda \mathbf{y}_i.e_i^{\mathbf{f}}, i \leq n\}$ using $\Delta_{\mathbf{f}}$.

The peephole hole rule relies on the fact that replacing a subexpression in the body of a definition by an expression that is equivalent independent of the interpretation of any function symbols that might appear.

Theorem (Peephole Rule): Let $\Delta_{\mathbf{g}}$ be obtained from $\Delta_{\mathbf{f}}$ by replacing some $F_i = C[e_0]$ by $G_i = C[e_1]\{\mathbf{f} := \lambda \mathbf{x}.\mathbf{g}(\mathbf{x})\}$. Suppose that $\theta \Rightarrow C[\![\theta_0]\!]$ and $\theta_0 \Rightarrow e_0 \cong e_1$ uniformly in $\mathbf{f}$, (i.e. established proof theoretically without using the unfolding rule for any f defined in $\Delta_{\mathbf{f}}$, or established semantically for arbitrary interpretations of the functions in $\mathbf{f}$). Then to prove

$$\theta \Rightarrow f_i(\mathbf{x}_i) \cong g_i(\mathbf{x}_i)$$

for $i \leq n$, it suffices to find a system of invariants $I = [(\phi_i, \psi_i), i \leq n]$ such that

- $\theta = \phi_i$, and
- $\Delta_{\mathbf{f}}$ is I-inductive.

We have chosen a simple instance of the peephole transformation so as not to get bogged down in notation. It is easy to generalize the peephole rule to allow for multiple replacements. Examples applications of the elaborated recursion induction principle and of the peephole rule can be found in the next section.

4.3 Principles for Introducing Memory Contexts

In this subsection we strengthen the recursion induction principle to allow for the introduction and manipulation of local memory context. For simplicity we focus on single cells contexts. We use c for the distinguished cell variable. Let L_v^c abbreviate $\mathtt{let}\{c := \mathtt{mk}(v)\}\bullet$. We narrow the scope of memory allocation via the memory allocation propagation relation, $L_v^c[e] \propto e'$. It is defined as follows.

(1) $L_v^c[e] \propto e$ if $c \notin \mathrm{FV}(e)$

(2) $L_v^c[\mathtt{get}(c)] \propto v$

(3) $L_v^c[\mathtt{seq}(\mathtt{set}(c, v'), c)] \propto \mathtt{mk}(v')$ if $c \notin \mathrm{FV}(v')$

(4) $L_v^c[\mathtt{if}(e_0, e_1, e_2)] \propto \mathtt{if}(e_0, e_1', e_2')$
if $c \notin \mathrm{FV}(e_0)$, and $L_v^c[e_j] \propto e_j'$ for $j \in \{1, 2\}$

(5) $L_v^c[\mathtt{let}\{x := e_0\}e_1] \propto \mathtt{let}\{c := e_0'\}\mathtt{let}\{y := \mathtt{get}(c)\}e_1'$
if $\mathtt{new}(c) \Rightarrow \mathtt{let}\{x := e_0\}[\![\mathtt{new}(c) \wedge c \cong x]\!]$, y is fresh,
$L_v^c[e_0] \propto e_0'$, and $L_y^c[e_1\{x := c\}] \propto e_1'$

(6) $L_v^c[\mathtt{let}\{x := e_0\}e_1] \propto \mathtt{let}\{x := e_0'\}e_1'$
if $\mathtt{new}(c) \wedge y \cong \mathtt{get}(c) \Rightarrow \mathtt{let}\{x := e_0\}[\![\mathtt{new}(c) \wedge c \; notin \, x \wedge \mathtt{get}(c) \cong y]\!]$
(with y is fresh), $L_v^c[e_0] \propto e_0'$, and $L_v^c[e_1] \propto e_1'$

(7) $L_v^c[e] \propto L_v^c[e]$ otherwise

The following simple fact is established by simple structural induction.

Lemma (M-prop):

(1) If $L_v^c[e] \propto e'$ then $L_v^c[e] \cong e'$.

(2) If $e^* = e\{x := \mathtt{get}(c)\}$ and $c \notin \mathrm{FV}(e)$, then $L_x^c[e^*] \propto e$.

Definition (*New*): To usefully propagate cell allocation into an expression it is necessary for the expression to propagate newness. For this purpose we define the system of invariants $New^c = [(\phi_i, \psi_i), i \leq n]$ where $\phi_i = \psi_i = \mathtt{new}(c)$ for $i \leq n$.

The memory reuse theorem provides a principle for carrying out a transformation within the context of a newly allocated memory cell.

Theorem (M-reuse): To prove

$$\phi_i \;\Rightarrow\; L_x^c[g_i(\mathbf{y}_i, c)] \cong f_i(\mathbf{y}_i, x)$$

for $i \leq n$, it suffices to show that there are expressions H_i, over $\mathbf{f}, \mathbf{y}_i, x$ for $i \leq n$, and there are systems of invariants $I^l = [(\phi_i, \psi_i^l), i \leq n]$ for $l < 2$ such that

(1) $\Delta_{\mathbf{f}}$ is I^0-inductive

(2) $\phi_i \;\Rightarrow\; L_x^c[g_i(\mathbf{y}_i, c)] \cong F_i\{f_j := \lambda \mathbf{y}_j, x.L_x^c[g_j(\mathbf{y}_j, c)] \;\big|\; j \leq n\}$, for $i \leq n$

(3) $\Delta_{\mathbf{g}}$ is New^c-inductive and $\{f_i(\mathbf{y}_i, x) \leftarrow H_i \;\big|\; i \leq n\}$ is I^1-inductive

(4) $L_x^c[G_i] \propto H_i\{f_j := \lambda \mathbf{y}_j, x.L_x^c[g_j(\mathbf{y}_j, c)] \;\big|\; i \leq n\}$, for $i \leq n$ (using the invariants proved for $\Delta_{\mathbf{g}}$).

(5) $\phi_i \;\Rightarrow\; f_i(\mathbf{y}_i, x) \cong H_i$, for $i \leq n$

The memory reuse principle is similar in spirit to recursion induction. Conditions (1,2) show that

$f_i \sqsubseteq \lambda \mathbf{y}_i, x.L_x^c[g_i(\mathbf{y}_i, c)]$, while conditions (3,4,5) show that

$\lambda \mathbf{y}_i, x.L_x^c[g_i(\mathbf{y}_i, c)] \sqsubseteq f_i$.

4.4 Register Passing Style Transform

Many compiler optimizations for functional languages, object systems, and imperative functional languages such as ML, Scheme, and Lisp, can be considered as program transformations in $\lambda_{\mathtt{mk}}$. One objective of the VTLoE work is to be able to justify and provide soundness criteria for such optimizations. Consider the following Scheme example, suggested by Felleisen [6]. Assume that `C` is a context with holes only in tail position, and that `A,B` are expressions, such that `loop` is not free in `A,B,C`, and both expressions below are closed. Then

```
(letrec ([loop (lambda (x) C[(loop A)])]) (loop B))
```

maybe safely replaced by

```
(letrec ([loop (lambda () C[(begin (set! x A) (loop))])] [x B]) (loop))
```

As a first simple application of our reasoning tools, we formalize a variant of this transformation in the first-order VTLoE setting. First we elaborate the

class $\mathbb{T}$ of contexts whose holes are all in *tail-recursive* position to allow indices on holes.

Definition ($\mathbb{T}$):

$$\mathbb{T} = \{\bullet_i\}_{i\in\mathbb{N}} + \mathbb{E} + \mathtt{let}\{\mathbb{X} := \mathbb{E}\}\mathbb{T} + \mathtt{if}(\mathbb{E}, \mathbb{T}, \mathbb{T})$$

We let T range over $\mathbb{T}$. We write $T[e_i]_i$ to mean that $\bullet_i$ is filled with e_i for i ranging over the indices of holes appearing in T. The general loop, `loop`, and its register passing variant, `rloop`, are defined by the following system.

$$\mathtt{loop}(x) \leftarrow T[\mathtt{loop}(e_i)]_i$$

$$\mathtt{rloop}(c) \leftarrow T^{\dagger}[\mathtt{seq}(\mathtt{set}(c, e_i^{\dagger}), \mathtt{rloop}(c))]_i$$

where $\mathtt{loop}, \mathtt{rloop}$ do not appear in $T[e_i]_i$, $T^{\dagger}$ abbreviates $T^{\{x:=\mathtt{get}(c)\}}$, and $e^{\dagger} = e^{\{x:=\mathtt{get}(c)\}}$. Note that the assumption of well-formedness of the definitions means that c is not free in T, or e_i.

Theorem (Reg-Passing): $\mathtt{loop}(x) \cong L_x^c[\mathtt{rloop}(c)]$

Proof : The proof is by (**m-reuse**) taking $n = 0$, $f_0 = \mathtt{loop}$, $g_0 = \mathtt{rloop}$, and ϕ_0 to be true. We take I^l to be the trivial invariant systems (all invariants true) for $l < 2$. To define H_0 we specialize the definition of $\propto$ to tail contexts with holes filled as in the definition of `rloop`. Thus we define

$$T^{\dagger}[\mathtt{let}\{y := e_i^{\dagger}\}\mathtt{seq}(\mathtt{set}(c, y), \mathtt{rloop}(c))]_i \propto H$$

by induction on the construction of T as follows. First note that

$$\mathtt{new}(c) \wedge x \cong \mathtt{get}(c) \Rightarrow \mathtt{let}\{w := e^{\dagger}\}[\![\mathtt{new}(c) \wedge c \mathit{\ notin\ } w \wedge x \cong \mathtt{get}(c)]\!]$$

for $e^{\dagger} = e\{x := \mathtt{get}(c)\}$ and $c \notin \mathrm{FV}(e)$. Thus (by (**mprop.2**)) $L_x^c[e^{\dagger}] \propto e$. The definition is according to the cases in the definition of T.

$T = \bullet_i \quad H = \mathtt{let}\{y := e_i\}L_y^c[\mathtt{rloop}(c)]$

$T = e$ (with no holes), then $H = e$

$T = \mathtt{if}(e_0, T_1, T_2)$, then $H = \mathtt{if}(e_0, H_1, H_2)$ where

$\quad T_j^{\dagger}[\mathtt{let}\{y := e_i^{\dagger}\}\mathtt{seq}(\mathtt{set}(c, y), \mathtt{rloop}(c))]_i \propto H_j$ for $j \in \{1, 2\}$

$T = \mathtt{let}\{x_0 := e_0\}T_1$, then $H = \mathtt{let}\{x_0 := e_0\}H_1$ where

$\quad T_1^{\dagger}[\mathtt{let}\{y := e_i^{\dagger}\}\mathtt{seq}(\mathtt{set}(c, y), \mathtt{rloop}(c))]_i \propto H_1$.

We take H_0 to be H with $L_y^c[\mathtt{rloop}(c)]$ replaced by $\mathtt{loop}(y)$.

Now we only need to establish the conditions (1-5) of (**m-reuse**).

(1) $\Delta_{\{\mathtt{loop}\}}$ is trivially I^0-inductive

(2) $L_x^c[\mathtt{rloop}(c)] \cong F_0\{\mathtt{loop} := \lambda x.L_x^c[\mathtt{rloop}(c)]\}$ since $L_x^c[\mathtt{rloop}(c)] \cong H$ and $H_0 \cong F_0$ using no defining equations.

(3) $\{\mathtt{loop}(x) \leftarrow H_0\}$ is trivially I^1-inductive. $\Delta_{\{\mathtt{rloop}\}}$ is New^c-inductive by the propagation property of $e^{\dagger}$ noted above.

(4) $L_x^c[G_0] \propto H$ by construction

(5) $\mathtt{loop}(x) \cong H_0$ trivially ($H_0 \cong F_0$ using no defining equations)

□

5 Transforming the Boyer Benchmark

The source of this example is a Common Lisp version of the well known Boyer benchmark (cf. [8]) that was used as the starting point for developing a parallel version [15]. The heart of the Boyer benchmark program is a rewrite rule based simplifier that repeatedly tries to rewrite a term with a given list of rules. The original Boyer benchmark program used lists and list operations to represent composite terms and lemmas. Variables and operations were represented as Lisp symbols. We represented atomic terms, composite terms, operations and lemmas as abstract structures using the Common Lisp `DEFSTRUCT` facility. A simple matcher decides if a rule applies. The matcher returns two pieces of information: a flag to indicate success or failure, and a substitution in case of success. The original version returned `t` or `nil` depending on success, and if successful set the value of a global variable to be the resulting substitution. We used the Common Lisp multiple values feature to accomplish this. We also used higher-order mapping functions instead of mutually recursive definitions in several places.

Although use of structure definitions, multiple values, and other high-level constructs results in elegant and easy to understand code, these constructs are part of a rather complex machinery and may well not produce the most efficient implementation of the underlying algorithm. The original program can (essentially) be recovered by carrying out the simple set of transformations described below.

(1) Rerepresent composite structures as lists and omit structure definitions for variables and operations.

(2) Eliminate the use of multiple values in favor of simple conses.

(3) Eliminate the use of high-order mapping constructs

(4) Eliminate unnecessary storage allocation due to multiple values by cell reuse.

In fact, the original application of these transformation rules was in the converse direction. That is we started with the original Boyer benchmark program an equivalent program that did not rely on global variables and assignment. This was essential in order to parallelize the program. It is likely that similar transformations from programs optimized for sequential computation to more abstract version will play an important role in developing parallel versions of existing programs.

The first three transformations apply quite generally and can be carried out automatically. The final transformation requires more insight and care, and we focus on that transformation here. We first transform the matcher, and then transform the rewriter to use the transformed matcher efficiently.

We use the following abbreviations to write programs in the usual vanilla

Lisp style:

$\texttt{not}(e)$	abbreviates	$\texttt{if}(e, \texttt{Nil}, \texttt{T})$
$\texttt{or}(e_0, e_1)$	abbreviates	$\texttt{if}(e_0, \texttt{T}, \texttt{if}(e_1, \texttt{T}, \texttt{Nil}))$
$\texttt{and}(e_0, e_1)$	abbreviates	$\texttt{if}(e_0, \texttt{if}(e_1, \texttt{T}, \texttt{Nil}), \texttt{Nil})$
$\texttt{null?}(x)$	abbreviates	$\texttt{eq?}(x, \texttt{Nil})$
$\texttt{cons}(x, y)$	abbreviates	$\texttt{pr}(\texttt{mk}(x, y))$
$\texttt{acons}(x, y, z)$	abbreviates	$\texttt{pr}(\texttt{pr}(x, y), z)$
$\texttt{car}(x)$	abbreviates	$\texttt{fst}(\texttt{get}(x))$
$\texttt{cdr}(x)$	abbreviates	$\texttt{snd}(\texttt{get}(x))$
$\texttt{setcar!}(x, y)$	abbreviates	$\texttt{seq}(\texttt{set}(x, \texttt{pr}(y, \texttt{cdr}(x))), x)$
$\texttt{setcdr!}(x, y)$	abbreviates	$\texttt{seq}(\texttt{set}(x, \texttt{pr}(\texttt{car}(x), y)), x)$
$\texttt{setpair!}(x, y, z)$	abbreviates	$\texttt{seq}(\texttt{set}(x, \texttt{pr}(y, z)), x)$

5.1 The matching transform

We begin by defining the basic data types to be used, as well as giving the definition of *match*, the abstract (specification) version of the matcher. Next we give an informal description of the transformation and the resulting final version of the program, and state the correctness of the transformation. Finally we outline the formal verification of the soundness of the transformation. This proves the correctness statement and provides additional information about the matcher.

A term is either an atomic term (variable) or a composite term obtained by application of an operation to a list of terms. Atomic terms and operations are just atoms, and application of an operation is represented by pairing it with the argument list. The abstract syntax of terms is thus represented as follows.

term-mk$(o, a) \leftarrow \texttt{pr}(o, a)$

term-op$(t) \leftarrow \texttt{fst}(t)$

term-args$(t) \leftarrow \texttt{snd}(t)$

A substitution is a finite map from variables to terms. Maps are represented as lists whose entries are pairs consisting of an atom and a term, i.e. as Lisp-like alists (our alists are immutable in contrast to the usual Lisp alist). Maps are extended using $\texttt{acons}(x, y, s)$ which abbreviates $\texttt{pr}(\texttt{pr}(x, y), s)$. A map is applied to an atom using *assoc* which returns the (first) pair whose `fst` is `eq?` to the atom, or `Nil` if no such pair exists. A substitution map is applied to a term using *app-sbst* which is defined as the obvious homomorphic extension of the substitution (mapping atoms not in the domain to themselves).

The specifying code for matching

The matcher *match* is given a pair of terms t_1, t_2 and a substitution s. The task of *match* is to determine whether or not t_1 is a substitution instance of t_2

via a substitution s' that extends s. *match* returns two pieces of information. Firstly, a boolean flag (`T` or `Nil`) indicating whether or not such a substitution exists, and secondly, when it exists, the substitution that achieves the match.

Definition (*match*):

```
match(t1, t2, s) ←
  cond([atom?(t2) ▷ let{b := assoc(t2, s)}
                      if(b,
                         if(term-eq(t1, snd(b)),
                            cons(T, s),
                            cons(Nil, Nil)),
                         cons(T, acons(t2, t1, s)))],
       [atom?(t1) ▷ cons(Nil, Nil)],
       [eq?(term-op(t1), term-op(t2))
            ▷ match-args(term-args(t1), term-args(t2), s)],
       [T ▷ cons(Nil, Nil)])

match-args(a1, a2, s) ← if(and(null?(a1), null?(a2)),
                           cons(T, s),
                           if(or(null?(a1), null?(a2)),
                              cons(Nil, Nil),
                              let{w := match(car(a1), car(a2), s)}
                                if(car(w),
                                   match-args(cdr(a1), cdr(a2), cdr(w)),
                                   cons(Nil, Nil))))
```

Transforming match

We observe that the result produced by *match* is a new cell and in recursive calls the cell returned by subcomputations is discarded. We will transform the definition of *match* to a version *match!* that takes a new cell initialized so that the `cdr` is the input substitution. This cell is reused and eventually returned as the result (with contents suitably updated). Thus `cons`es constructing the result are replaced by `setpair`s on the reusable cell, and third argument to the call to *match!* from *match-args!* is now 'by reference'. The resulting definition of *match!* is the following.

Definition (*match!*):

```
match!(t1, t2, c) ←
```

$$
\begin{array}{l}
\texttt{cond}([\texttt{atom?}(t_2) \rhd \texttt{let}\{b := \mathit{assoc}(t_2, \texttt{cdr}(c))\} \\
\qquad\qquad\qquad\qquad \texttt{if}(b, \\
\qquad\qquad\qquad\qquad\quad \texttt{if}(\mathit{term\text{-}eq}(t_1, \texttt{snd}(b)), \\
\qquad\qquad\qquad\qquad\qquad \texttt{setcar!}(c, \texttt{T}), \\
\qquad\qquad\qquad\qquad\qquad \texttt{setpair!}(c, \texttt{Nil}, \texttt{Nil})), \\
\qquad\qquad\qquad\qquad\quad \texttt{setpair!}(c, \texttt{T}, \texttt{acons}(t_2, t_1, \texttt{cdr}(c))))], \\
\qquad [\texttt{atom?}(t_1) \rhd \texttt{setpair!}(c, \texttt{Nil}, \texttt{Nil})], \\
\qquad [\texttt{eq?}(\mathit{term\text{-}op}(t_1), \mathit{term\text{-}op}(t_2)) \\
\qquad\qquad \rhd \mathit{match\text{-}args!}(\mathit{term\text{-}args}(t_1), \mathit{term\text{-}args}(t_2), c)], \\
\qquad [\texttt{T} \rhd \texttt{setpair!}(c, \texttt{Nil}, \texttt{Nil})])
\end{array}
$$

$$
\begin{array}{l}
\mathit{match\text{-}args!}(a_1, a_2, c) \leftarrow \texttt{if}(\texttt{and}(\texttt{null?}(a_1), \texttt{null?}(a_2)), \\
\qquad\qquad\qquad\qquad \texttt{setcar!}(c, \texttt{T}), \\
\qquad\qquad\qquad\qquad \texttt{if}(\texttt{or}(\texttt{null?}(a_1), \texttt{null?}(a_2)), \\
\qquad\qquad\qquad\qquad\quad \texttt{setpair!}(c, \texttt{Nil}, \texttt{Nil}), \\
\qquad\qquad\qquad\qquad\quad \texttt{seq}(\mathit{match!}(\texttt{car}(a_1), \texttt{car}(a_2), c), \\
\qquad\qquad\qquad\qquad\qquad \texttt{if}(\texttt{car}(c), \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{match\text{-}args!}(\texttt{cdr}(a_1), \texttt{cdr}(a_2), c), \\
\qquad\qquad\qquad\qquad\qquad\quad \texttt{setpair!}(c, \texttt{Nil}, \texttt{Nil})))))
\end{array}
$$

Although the intended domain of *match* constrains the first two arguments to be terms and the third argument to be a substitution, the transformation preserves equivalence under more general conditions.

Theorem (match):

$$\mathit{match}(t_1, t_2, s) \cong \texttt{let}\{c := \texttt{cons}(x, s)\}\mathit{match!}(t_1, t_2, c)$$

Proof (match): The proof of (**match**) uses (**m-reuse**). In order to use this principle as stated, we need to modify *match*, *match-args* to take a pair as their third argument, and ignore the first component. We show the modified form is equivalent to the original. We then establish the key invariants for *match!*, *match-args!*. Finally we use (**m-reuse**) to establish the equivalence of the modified matcher to the fully transformed matcher. Thus (**match**) follows directly from (**modified match**) and (**match'**) below. $\square_{\textbf{match}}$

We will use the following notation. *M*, *Ma* are the bodies of the defining equations for *match*, and *match-args* respectively. *M!*, *Ma!* are the bodies of the defining equations for *match!*, and *match-args!* respectively.

Definition (*match'*, *match-args'*): Let *match'*, *match-args'* be the modified matcher with defining equations

$$\mathit{match'}(t_1, t_2, y) \leftarrow M'$$

$$\mathit{match\text{-}args'}(a_1, a_2, y) \leftarrow Ma'$$

where M' is obtained from M by replacing: *match-args* by *match-args'*; the first three occurrences of s by $\mathtt{snd}(y)$; and the final occurrence (in the call to *match-args'*) by y. Ma' is obtained from Ma by replacing: *match* by *match'*; *match-args* by *match-args'*; the first occurrence of s by $\mathtt{snd}(y)$; and the second occurrence (in the call to *match'*) by y; and the occurrence of $\mathtt{cdr}(w)$ (in the call to *match-args'*) by $\mathtt{pr}(\mathtt{fst}(y), \mathtt{cdr}(w))$.

Lemma (modified match):

$$\mathtt{pr?}(y) \cong \mathtt{T} \wedge \mathtt{snd}(y) \cong s \Rightarrow$$
$$\mathit{match}(t_1, t_2, s) \cong \mathit{match'}(t_1, t_2, y) \wedge$$
$$\mathit{match\text{-}args}(a_1, a_2, s) \cong \mathit{match\text{-}args'}(a_1, a_2, y)$$

Proof : This is proved by (elaborated) recursion induction, taking

$$e_{\mathit{match}} = \mathit{match}(t_1, t_2, \mathtt{snd}(y))$$
$$e_{\mathit{match\text{-}args}} = \mathit{match\text{-}args}(a_1, a_2, \mathtt{snd}(y))$$
$$e_{\mathit{match'}} = \mathit{match'}(t_1, t_2, \mathtt{pr}(\mathtt{fst}(y), s))$$
$$e_{\mathit{match\text{-}args'}} = \mathit{match\text{-}args'}(a_1, a_2, \mathtt{pr}(\mathtt{fst}(y), s))$$

□

The key invariants for *match!* and *match-args!* are preservation of newness of the third argument, and the return of that argument. This is expressed in the following lemma.

Lemma (match-new): Let $\phi = \mathtt{new}(c) \wedge \mathtt{pr?}(\mathtt{get}(c)) \cong \mathtt{T}$, and let $\psi = \phi \wedge z \cong c$, then

$$\mathtt{new}(c) \Rightarrow \mathtt{seq}(\mathit{match!}(t_1, t_2, c), [\![\mathtt{new}(c)]\!])$$
$$\mathtt{new}(c) \Rightarrow \mathtt{seq}(\mathit{match\text{-}args!}(a_1, a_2, c), [\![\mathtt{new}(c)]\!])$$
$$\phi \Rightarrow \mathtt{let}\{z := \mathit{match!}(t_1, t_2, c)\}[\![\psi]\!]$$
$$\phi \Rightarrow \mathtt{let}\{z := \mathit{match\text{-}args!}(a_1, a_2, c)\}[\![\psi]\!]$$

Proof : By (**subgoal induction**) using an argument similar to that found in the proof of (**sum!**) in the previous section. □

Lemma (match'):

$$\mathit{match'}(t_1, t_2, y) \cong L^c_y[\mathit{match!}(t_1, t_2, c)]$$
$$\mathit{match\text{-}args'}(a_1, a_2, y) \cong L^c_y[\mathit{match\text{-}args!}(a_1, a_2, c)]$$

Proof : By (**m-reuse**). To see this, define H, Ha as follows: H is M'; and Ha is Ma' with the subexpression

$$\mathtt{let}\{w := \mathit{match}(\mathtt{car}(a_1), \mathtt{car}(a_2), s)\}\mathtt{if}(\mathtt{car}(w), \ldots, \ldots)$$

replaced by

$$\texttt{let}\{w := \mathit{match}(\texttt{car}(a_1), \texttt{car}(a_2), s)\}\texttt{let}\{p := \texttt{get}(w)\}\texttt{if}(\texttt{fst}(p), \ldots, \ldots).$$

Let Σ be the function substitution list

$$\mathit{match}' := \lambda t_1, t_2, y.L_y^c\, \mathit{match!}(t_1, t_2, c),$$

$$\mathit{match\text{-}args}' := \lambda a_1, a_2, y.L_y^c\, \mathit{match\text{-}args!}(a_1, a_2, c).$$

Then, by (**match-new**), the main work that remains is to show

(a) $L_y^c[M!] \propto H\{\Sigma\}$, and $L_y^c[Ma!] \propto Ha\{\Sigma\}$

(b) $L_y^c[\mathit{match!}(t_1, t_2, c)] \cong M'\{\Sigma\}$, and $L_y^c[\mathit{match\text{-}args!}(a_1, a_2, c)] \cong Ma'\{\Sigma\}$

(c) $\mathit{match}'(t_1, t_2, y) \cong H$, and $\mathit{match\text{-}args}'(a_1, a_2, y) \cong Ha$

(b) follows from (a) and (**m-prop**), (c) can easily be established using the Peephole rule. To establish (a) we use the definition of $\propto$. There are only three interesting points here. The first is the propagation of L_y^c across the subexpression $\texttt{let}\{b := \mathit{assoc}(t_2, \texttt{cdr}(c)\}\ldots$ in *match!*. Using (d) below we may apply clause (6) of the definition of $\propto$.

(d) $\texttt{new}(c) \wedge v \cong \texttt{get}(c) \Rightarrow$

$$\texttt{let}\{b := \mathit{assoc}(t_2, \texttt{cdr}(c))\}[\![\texttt{new}(c) \wedge v \cong \texttt{get}(c) \wedge c \mathit{\ notin\ } b]\!]$$

(d) is easy to establish using the definition of **new** and the fact that *assoc* is purely functional.

The second point is the propagation of L_y^c across the subexpression

$$\texttt{seq}(\mathit{match!}(\texttt{car}(a_1), \texttt{car}(a_2), c), \ldots)$$

in *match-args!*. We expand the **seq**, replacing this subexpression by

$$\texttt{let}\{d := \mathit{match!}(\texttt{car}(a_1), \texttt{car}(a_2), c)\}\ldots).$$

Now, by (**match-new**) we may apply clause (5) of the definition of $\propto$.

The third point is the propagation of L_y^c across the subexpression

$$\texttt{if}(\texttt{car}(c), \ldots, \ldots).$$

We rewrite this (using the Peephole rule) to

$$\texttt{let}\{p := \texttt{get}(c)\}\texttt{if}(\texttt{fst}(p), \ldots, \ldots),$$

then using (e) below, we may again apply clause (6).

(e) $\texttt{new}(c) \wedge v \cong \texttt{get}(c) \Rightarrow$

$$\texttt{let}\{b := \texttt{get}(c)\}[\![\texttt{new}(c) \wedge v \cong \texttt{get}(c) \wedge c \mathit{\ notin\ } b]\!]$$

□

5.2 Transforming the rewriter

We extend the abstract syntax for matching. A *lemma* has a left-hand side and a right-hand side, each of which is a term.

$$\textit{lemma-lhs}(x) \leftarrow \texttt{fst}(x)$$
$$\textit{lemma-rhs}(x) \leftarrow \texttt{snd}(x)$$

A lemma corresponds to an equation to be used as a rewrite rule in the left-to-right direction.

The specification code

The rewriter *rewrite* takes as input a term t and a list of lemmas l. If the term is atomic the program exits with that term as its value. Otherwise the term is a composite term consisting of an operation and an list of argument terms. The argument subterms are first rewritten and then the top-level rewriter *rewrite-top* is applied to the resulting whole term. The top-level rewriter applies the first lemma that matches (if any) and then restarts the rewriting process. This is repeated until no more rewriting can be done.

Definition (*rewrite*):

$$\begin{aligned}
&\textit{rewrite}(t, l) \leftarrow \texttt{if}(\texttt{atom?}(t),\\
&\qquad\qquad t,\\
&\qquad\qquad \texttt{let}\{t' := \textit{term-mk}(\textit{term-op}(t),\\
&\qquad\qquad\qquad\qquad\qquad \textit{rewrite-args}(\textit{term-args}(t), l))\}\\
&\qquad\quad \textit{rewrite-top}(t', l, l))\\
&\textit{rewrite-args}(a, l) \leftarrow \texttt{if}(\texttt{null?}(a),\\
&\qquad\qquad\quad \texttt{Nil},\\
&\qquad\qquad\quad \texttt{pr}(\textit{rewrite}(\texttt{fst}(a), l), \textit{rewrite-args}(\texttt{snd}(a), l))\\
&\textit{rewrite-top}(t, cl, l) \leftarrow\\
&\quad \texttt{if}(\texttt{null?}(cl),\\
&\qquad t,\\
&\qquad \texttt{let}\{w := \textit{match}(t, \textit{lemma-lhs}(\texttt{car}(cl)), \texttt{Nil})\}\\
&\qquad\quad \texttt{if}(\texttt{car}(w),\\
&\qquad\qquad \textit{rewrite}(\textit{app-sbst}(\texttt{cdr}(w), \textit{lemma-rhs}(\texttt{car}(cl))), l),\\
&\qquad\qquad \textit{rewrite-top}(t, \texttt{cdr}(cl), l)))
\end{aligned}$$

Transformation steps

We transform *rewrite* into *rewrite!* just as we transformed *match* into *match!*.

Definition (*rewrite!*)**:**

$$
\begin{aligned}
&\mathit{rewrite!}(t, l, c) \leftarrow \mathtt{if}(\mathtt{atom?}(t),\\
&\qquad\qquad t,\\
&\qquad\qquad \mathtt{let}\{t' := \mathit{term\text{-}mk}(\mathit{term\text{-}op}(t),\\
&\qquad\qquad\qquad\qquad \mathit{rewrite\text{-}args!}(\mathit{term\text{-}args}(t), l, c))\}\\
&\qquad\qquad \mathit{rewrite\text{-}top!}(t', l, l, c))\\
&\mathit{rewrite\text{-}args!}(a, l, c) \leftarrow \mathtt{if}(\mathtt{null?}(a),\\
&\qquad\qquad\qquad \mathtt{Nil},\\
&\qquad\qquad\qquad \mathtt{pr}(\mathit{rewrite!}(\mathtt{fst}(a), l, c),\\
&\qquad\qquad\qquad\quad \mathit{rewrite\text{-}args!}(\mathtt{snd}(a), l, c))\\
&\mathit{rewrite\text{-}top}(t, cl, l, c) \leftarrow\\
&\quad \mathtt{if}(\mathtt{null?}(cl),\\
&\qquad t,\\
&\qquad \mathtt{seq}(\mathtt{setpair!}(c, \mathtt{T}, \mathtt{Nil}),\\
&\qquad\quad \mathit{match!}(t, \mathit{lemma\text{-}lhs}(\mathtt{car}(cl)), c)\}\\
&\qquad\quad \mathtt{if}(\mathtt{car}(c),\\
&\qquad\qquad \mathit{rewrite!}(\mathit{app\text{-}sbst}(\mathtt{cdr}(c), \mathit{lemma\text{-}rhs}(\mathtt{car}(cl))), l, c),\\
&\qquad\qquad \mathit{rewrite\text{-}top}(t, \mathtt{cdr}(cl), l, c)))
\end{aligned}
$$

The validity of the transformation is expressed by the following theorem.

Theorem (rewrite):

(rw) $\mathit{rewrite}(t, l) \cong \mathtt{let}\{c := \mathtt{cons}(\mathtt{T}, \mathtt{Nil})\}\mathit{rewrite!}(t, l, c)$

(rwa) $\mathit{rewrite\text{-}args}(a, l) \cong \mathtt{let}\{c := \mathtt{cons}(\mathtt{T}, \mathtt{Nil})\}\mathit{rewrite\text{-}args!}(a, l, c)$

(rwt) $\mathit{rewrite\text{-}top}(t, cl, l) \cong \mathtt{let}\{c := \mathtt{cons}(\mathtt{T}, \mathtt{Nil})\}\mathit{rewrite\text{-}top!}(t, cl, l, c)$

6 Conclusion

In this paper we have presented rules for systematically transforming programs in the first-order fragment of a Lisp- (Scheme-, ML-) like language with first-order recursive function definitions and objects with memory. The validity of the transformations is based on a theory of constrained equivalence and contextual assertions. Subgoal and recursion induction are standard methods of proving properties of flow-chart and first-order programs acting on immutable data [21, 25]. We have generalized these methods to programs that act on mutable data structures. The contextual assertion notation allows one to express notions such as *exclusive use* which allow one to replace allocation by updating [12].

Even though the work presented here is at an early stage of development, we feel that much progress has been made towards practical application of formal

methods to program development. Work is in progress to apply these methods to more substantial programming examples that arise in practice, and extend the methods to the full $\lambda_{\mathtt{mk}}$-calculus.

For the transformational approach we have focused on methods for proving equivalence of definitions, as well as establishing invariants (partial correctness statements) needed to prove equivalence. For this purpose subgoal induction and recursion induction are appropriate. Treatment of total correctness requires the formulation of principles for induction on well-founded orderings. Examples of structural induction principles extended to the case of computations with effects are given in [19] and additional principles have been formulated to treat cases where the measure that is being decreased is not a simple structural property, but decreases due the effect that is produced by the computation. In [10] classes are used to express various forms of induction.

The atomic formula new treats the simplest case of a newly allocated structure. Another example to which we have applied our methods is the derivation of space efficient programs for manipulating polynomials. This requires formalizing the notion of newly allocated polynomial structures, either as a subtype of lists, or as a mutable abstract data type. Future work will extend the techniques presented in this paper to general mutable and immutable abstract data types.

Acknowledgements

We would like to thank Anna Patterson, Morten Heine Sørensen, and Scott Smith for helpful comments on an earlier draft. This research was partially supported by DARPA contract NAG2-703, NSF grants CCR-8917606, and CCR-8915663.

References

1. K.R. Apt. Ten years of Hoare's logic: A survey–part I. *ACM Transactions on Programming Languages and Systems*, 4:431–483, 1981.
2. S. Feferman. A language and axioms for explicit mathematics. In *Algebra and Logic*, volume 450 of *Springer Lecture Notes in Mathematics*, pages 87–139. Springer Verlag, 1975.
3. S. Feferman. Constructive theories of functions and classes. In *Logic Colloquium '78*, pages 159–224. North-Holland, 1979.
4. S. Feferman. A theory of variable types. *Revista Colombiana de Matématicas*, 19:95–105, 1985.
5. S. Feferman. Polymorphic typed lambda-calculi in a type-free axiomatic framework. In *Logic and Computation*, volume 106 of *Contemporary Mathematics*, pages 101–136. A.M.S., Providence R. I., 1990.
6. M. Felleisen, 1993. Personal communication.
7. M. Felleisen and D.P. Friedman. Control operators, the SECD-machine, and the λ-calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217. North-Holland, 1986.
8. Richard P. Gabriel. *Performance and Evaluation of Lisp Systems*. Computer Systems Series. MIT Press, Cambridge, Massachusetts, 1985.

9. D. Harel. Dynamic logic. In D. Gabbay and G. Guenthner, editors, *Handbook of Philosophical Logic, Vol. II*, pages 497–604. D. Reidel, 1984.
10. F. Honsell, I. A. Mason, S. F. Smith, and C. L. Talcott. A Variable Typed Logic of Effects. *Information and Computation*, ???(???):???–???, 199?
11. F. Honsell, I. A. Mason, S. F. Smith, and C. L. Talcott. A theory of classes for a functional language with effects. In *Proceedings of CSL92*, volume ??? of *Lecture Notes in Computer Science*, pages ???–??? Springer, Berlin, 1993.
12. U. Jø rring and W. L. Scherlis. Deriving and using destructive data types. In *IFIP TC2 Working Conference on Program Specification and Transformation*. North–Holland, 1986.
13. I. A. Mason. *The Semantics of Destructive Lisp*. PhD thesis, Stanford University, 1986. Also available as CSLI Lecture Notes No. 5, Center for the Study of Language and Information, Stanford University.
14. I. A. Mason. Verification of programs that destructively manipulate data. *Science of Computer Programming*, 10:177–210, 1988.
15. I. A. Mason, J. D. Pehoushek, C. L. Talcott, and J. S. Weening. A Qlisp Primer. Technical Report STAN-CS-90-1340, Department of Computer Science, Stanford University, 1990.
16. I. A. Mason and C. L. Talcott. Axiomatizing operational equivalence in the presence of side effects. In *Fourth Annual Symposium on Logic in Computer Science*. IEEE, 1989.
17. I. A. Mason and C. L. Talcott. Programming, transforming, and proving with function abstractions and memories. In *Proceedings of the 16th EATCS Colloquium on Automata, Languages, and Programming, Stresa*, volume 372 of *Lecture Notes in Computer Science*, pages 574–588. Springer-Verlag, 1989.
18. I. A. Mason and C. L. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1:287–327, 1991.
19. I. A. Mason and C. L. Talcott. Inferring the equivalence of functional programs that mutate data. *Theoretical Computer Science*, 105(2):167–215, 1992.
20. I. A. Mason and C. L. Talcott. References, local variables and operational reasoning. In *Seventh Annual Symposium on Logic in Computer Science*, pages 186–197. IEEE, 1992.
21. J. McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D Herschberg, editors, *Computer programming and formal systems*, pages 33–70. North-Holland, 1963.
22. E. Moggi. Computational lambda-calculus and monads. Technical Report ECS-LFCS-88-86, University of Edinburgh, 1988.
23. E. Moggi. Computational lambda-calculus and monads. In *Fourth Annual Symposium on Logic in Computer Science*. IEEE, 1989.
24. J. H. Morris. *Lambda calculus models of programming languages*. PhD thesis, Massachusetts Institute of Technology, 1968.
25. J. H. Morris and B. Wegbreit. Subgoal induction. *Communications of the Association for Computing Machinery*, 20:209–222, 1976.
26. G. Plotkin. Call-by-name, call-by-value and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.
27. J.C. Reynolds. Idealized ALGOL and its specification logic. In D. Néel, editor, *Tools and Notions for Program Construction*, pages 121–161. Cambridge University Press, 1982.

This article was processed using the LaTeX macro package with LLNCS style