Program Transformation via Constraint Propagation

Ian Mason
Stanford University
IAM@SAIL.STANFORD.EDU

Carolyn Talcott
Stanford University
CLT@SAIL.STANFORD.EDU

1. Introduction

Real programs have effects—creating new structures, examining and modifying existing structures, altering flow of control, etc. Such facilities are important not only for optimization, but also for communication, clarity, and simplicity in programming. Thus it is important to be able to reason both informally and formally about programs with effects, and not to sweep effects either to the side or under the store parameter rug. In this paper we describe progress towards a theory of program development by systematic refinement beginning with a clean, functional 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/ML). Such notions of equivalence are fundamental for the process of program specification, derivation, and transformation, as well as other forms of code generation and optimization.

The cornerstone of our theory 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 [12] and the lambda-c calculus [9]. The theory of operational equivalence for a higher-order functional/imperative language is presented in [6].

Constrained equivalence is a relation between sets of constraints (on computation contexts) and pairs of expressions. The interpretation is that in all contexts satisfying the constraints, evaluation of the expressions is either undefined or produces the same results and has the same effect on memory (modulo garbage). Constrained equivalence naturally allows reasoning by cases and permits use of a variety of induction principles. An inference system for constrained equivalence that is complete for expressions not involving recursively defined functions is given in [5, 7].

One difficulty with constrained equivalence is that it is not a congruence. One cannot, in general, place expressions equivalent under some non-empty set of constraints into an arbitrary program context and preserve equivalence. For this and other reasons, we have developed a formal system for propagating constraints into program contexts. In this system, it is possible to place expressions equivalent under some non-empty set of constraints into a program context and preserve equivalence provided that the constraints propagate into that context. Constrained equivalence and constraint propagation provide a basis for systematic development of program transformation rules. In this paper we present three main rules: subgoal induction, recursion induction, and the peephole rule. Use of these rules is illustrated by the transformation of a simple, but non-trivial, rewriting program, known as the Boyer Benchmark [1]. The example is part of a more extensive transformational development carried out in the process of developing parallel Lisp programs for symbolic manipulation [4]. The peephole rule answers affirmatively a conjecture in [3], p 178.

The remainder of this paper is organized as follows. In section 2. we present the basic framework. In section 3. we describe the formal system for constrained equivalence and constraint propagation. In section 4. we present the main transformation rules. In section 5. these rules are used to transform the Boyer Benchmark. Section 6. contains concluding remarks and directions for future work. For completeness the full set of rules for constrained equivalence and constraint propagation are given in two appendices.

We conclude this section with a summary of notational conventions. We use the usual notation for set membership and function application. Let Y, Y_0, Y_1 be sets. 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. $\mathbf{P}_\omega(Y)$ is the set of finite subsets of Y. $[Y_0 \to Y_1]$ 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. 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 Framework

Our language can be thought of as untyped ML (with binary rather than unary memory cells), or as a variant of Scheme in which naming of values and memory allocation have been separated. Thus there are explicit memory operations (*cell*, *eq*, *cons*, *car*, *setcar*, etc.) 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. The memory operations are interpreted relative to a given memory as follows. *cell* is the characteristic function of the cells, using the booleans `T` and `Nil`; *eq* tests whether two values are identical. *cons* takes two arguments, creates a new cell (extending the memory domain) whose contents is the pair of arguments, and returns the newly created cell. *car* and *cdr* return the first and second components of a cell. *setcar* and *setcdr* destructively alter an already existing cell. Given two arguments, the first of which must be a cell, *setcar* updates the given memory so that the first component of the contents of its first argument becomes its second argument. *setcdr* similarly alters the second component. Thus memories can be constructed in which one or both components of a cell can refer to the cell itself. For simplicity of presentation, we focus on the first-order fragment in this paper.

2.1. The language

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, $\mathcal{F}_n$, of n-ary function symbols. We assume the sets $\mathbb{A}$, $\mathbb{X}$, and $\mathcal{F}_n$ for $n \in \mathbb{N}$ are pairwise disjoint. We let $\mathbb{F}_n$ denote the set of n-ary memory operations where the unary memory operations are $\mathbb{F}_1 = \{cell, car, cdr\}$ and the binary memory operations are $\mathbb{F}_2 = \{eq, cons, setcar, setcdr\}$. We let $\mathfrak{F}_n$ abbreviate $\mathbb{F}_n \cup \mathcal{F}_n$ and $\mathfrak{F}$ abbreviates $\bigcup_{n \in \mathbb{N}} \mathfrak{F}_n$.

Definition ($\mathbb{U}$ $\mathbb{E}$): The set of value expressions, $\mathbb{U}$, and the set of expressions, $\mathbb{E}$, are defined to be the least sets satisfying the following equations:

$$\mathbb{U} = \mathbb{X} \cup \mathbb{A}$$

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

We let $a, a_0, \ldots$ range over $\mathbb{A}$, $x, x_0, y, z, \ldots$ range over $\mathbb{X}$, $u, u_0, \ldots$ range over $\mathbb{U}$, and $e, e_0, \ldots$ 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'. A value substitution is a finite map from variables to value expressions. We let σ, σ_0, ...range over value substitutions. 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.

Contexts are expressions with holes. We use ε to denote a hole. We let C, C' range over ${}^{\varepsilon}\mathbb{E}$. $C[\![e]\!]$ denotes the result of replacing any holes in C by e. Free variables of e may become bound in this process.

Definition ($\mathfrak{D}$): The set of (recursive function-) definition sets $\mathfrak{D}$ is the collection of all finite sequences of $n+1$-tuples of the form $<\mathrm{f}(x_1, \ldots, x_n) \leftarrow e>$ where f is an n-ary function symbol.

$$\mathfrak{D} = (\bigcup_{n \in \mathbb{N}} <\mathcal{F}_n(\mathbb{X}^n) \leftarrow \mathbb{E}>)^*$$

Let $\mathcal{D} \in \mathfrak{D}$ be a definition set. The defined functions of $\mathcal{D}$ are those $\mathrm{f} \in \mathcal{F}$ for which there is a definition $<\mathrm{f}(x_1, \ldots, x_n) \leftarrow e>$ occurring in $\mathcal{D}$ 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 set $\mathcal{D}$ is well-formed if no function symbol is defined more than once, and for each $<\mathrm{f}(x_1, \ldots, x_n) \leftarrow e>$ in $\mathcal{D}$, 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 $\mathcal{D}$. We shall assume that definition sets are well-formed unless otherwise stated. Within a single definition the formal parameters are bound in the body and we may freely rename (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 $\mathtt{if}(e_0, \mathtt{seq}(e_1, \ldots, e_n), \mathtt{seq}(e_1, \ldots, e_n))$

$\mathtt{cond}()$ abbreviates $\mathtt{Nil}$

$\mathtt{cond}([e_0 \Rightarrow e_0'], [e_1 \Rightarrow e_1'], \ldots, [e_n \Rightarrow e_n'])$

abbreviates $\mathtt{if}(e_0, e_0', \mathtt{cond}([e_1 \Rightarrow e_1'], \ldots, [e_n \Rightarrow e_n']))$

$\mathtt{not}(e)$ abbreviates $\mathtt{if}(e, \mathtt{Nil}, \mathtt{T})$

$\mathtt{or}(e_0, e_1)$ abbreviates $\mathtt{if}(e_0, \mathtt{T}, \mathtt{if}(e_1, \mathtt{T}, qnil))$

$\mathtt{and}(e_0, e_1)$ abbreviates $\mathtt{if}(e_0, \mathtt{if}(e_1, \mathtt{T}, qnil), \mathtt{Nil})$

$null(x)$ abbreviates $eq(x, \mathtt{Nil})$

$atom(x)$ abbreviates $\mathtt{not}(cell(x))$

$setpair(x, y, z)$ abbreviates $\mathtt{seq}(setcar(x, y), setcdr(x, z), x)$

2.2. The semantics

In [6] we provided two operational semantics. The first, a standard operational semantics, was based on memory structures, while the second was based on a syntactic reduction to normal form. We outline the reduction semantics for the first-order fragment here, as it provides a natural basis for reasoning about program equivalence.

The operational semantics of expressions relative to a definition set $\mathcal{D}$ 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. The set of redexes, $\mathbb{E}_{\mathtt{rdx}}$, is defined as

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

An expression is either a value expression or decomposes uniquely into a primitive expression placed in a reduction context. Reduction contexts identify the subexpression of an expression that is to be evaluated next. The set of reduction contexts, $\mathbb{R}$, is the subset of ${}^{\varepsilon}\mathbb{E}$ defined by

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

We let R range over $\mathbb{R}$. We often adopt the usual convention that $[\![\ ]\!]$ denotes a hole. Another important class of contexts are memory contexts, a syntactic representation of memory state. They are contexts Γ of the form:

$$\begin{aligned}&\mathtt{let}\{z_1 := cons(\mathtt{Nil}, \mathtt{Nil})\} \ldots \mathtt{let}\{z_n := cons(\mathtt{Nil}, \mathtt{Nil})\} \\ &\quad \mathtt{seq}(setcar(z_1, u_1^a), setcdr(z_1, u_1^d), \ldots, setcar(z_n, u_n^a), setcdr(z_n, u_n^d), \varepsilon)\end{aligned}$$

where $z_i \neq z_j$ when $i \neq j$. Memory contexts are the syntactic analog of a memory. As descriptions of memories we can view them as finite maps from variables to pairs of value expressions. We define $\mathrm{Dom}(\Gamma) = \{z_1, \ldots, z_n\}$ and $\Gamma(z_i) = [u_i^a, u_i^d]$ for $1 \leq i \leq n$. $\Gamma\{z := [u_a, u_d]\}$ is the memory context Γ' such that $\mathrm{Dom}(\Gamma') = \mathrm{Dom}(\Gamma) \cup \{z\}$, $\Gamma'(z) = [u_a, u_d]$, and $\Gamma'(z') = \Gamma(z')$ if $z \neq z'$.

Single-step reduction ($\mapsto$) is a relation on pairs $\Gamma; e$ consisting of a memory context and an expression, with $\mathrm{FV}(e) \subseteq \mathrm{Dom}(\Gamma)$. We call such pairs *descriptions*. The reduction relation $\overset{*}{\mapsto}$ is the reflexive transitive closure of $\mapsto$. A *value description* is a memory context together with a value expression. A description is *defined* just if it reduces, via $\overset{*}{\mapsto}$ to a value description. When expressions are used where descriptions are expected we take the memory context to be empty. Officially, $\overset{*}{\mapsto}$ and $\mapsto$ should be parameterized by the definition set $\mathcal{D}$, we will not make this parameter explicit here. Single-step reduction is the least relation such that

$$\text{(let)}\qquad \Gamma; R[\![\mathtt{let}\{x := u\}e]\!] \mapsto \Gamma; R[\![e\{x := u\}]\!]$$

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

$$\text{(rec)}\qquad \Gamma; R[\![\mathrm{f}(u_1, \ldots, u_n)]\!] \mapsto \Gamma; R[\![e^{\sigma}]\!]$$

$$\text{if } \mathrm{<f}(x_1, \ldots, x_n) \leftarrow e\mathrm{>} \text{ in } \mathcal{D},\ \mathrm{Dom}(\sigma) = \{x_1, \ldots, x_n\} \text{ and } \sigma(x_i) = u_i \text{ for } 1 \le i \le n$$

$$\text{(delta)}\qquad \Gamma; R[\![\mathrm{f}(u_1, \ldots, u_n)]\!] \mapsto \Gamma'; R[\![u']\!]$$

where in (**delta**) we assume that $\Gamma; R[\![\mathrm{f}(u_1, \ldots, u_n)]\!] \xrightarrow{\mathrm{p}} \Gamma'; R[\![u']\!]$ and, for example,

$$\Gamma; R[\![\mathit{cell}(u)]\!] \xrightarrow{\mathrm{p}} \begin{cases} \Gamma; R[\![\mathtt{T}]\!] & \text{if } u \in \mathrm{Dom}(\Gamma) \\ \Gamma; R[\![\mathtt{Nil}]\!] & \text{otherwise} \end{cases}$$

$$\Gamma; R[\![\mathit{eq}(u_0, u_1)]\!] \xrightarrow{\mathrm{p}} \begin{cases} \Gamma; R[\![\mathtt{T}]\!] & \text{if } u_0 = u_1 \text{ and } u_i \in \mathbb{A} \cup \mathrm{Dom}(\Gamma) \text{ for } i < 2 \\ \Gamma; R[\![\mathtt{Nil}]\!] & \text{otherwise} \end{cases}$$

$$\Gamma; R[\![\mathit{cons}(u_0, u_1)]\!] \xrightarrow{\mathrm{p}} \Gamma\{z := [u_0, u_1]\}; R[\![z]\!]$$

$$\Gamma; R[\![\mathit{car}(z)]\!] \xrightarrow{\mathrm{p}} \Gamma; R[\![u_a]\!]$$

$$\Gamma; R[\![\mathit{setcar}(z, u)]\!] \xrightarrow{\mathrm{p}} \Gamma\{z := [u, u_d]\}; R[\![z]\!]$$

where in the *cons* rule $z \notin \mathrm{Dom}(\Gamma) \cup \mathrm{FV}(R[\![u_i]\!])$, $i \le 2$, and in the *car*, *cdr*, *setcar*, and *setcdr* rules we assume $z \in \mathrm{Dom}(\Gamma)$ and $\Gamma(z) = [u_a, u_d]$.

2.3. Notions of equivalence

Now we define two notions of program equivalence: operational equivalence and constrained equivalence.

Definition ($\cong$): Two expressions are operationally equivalent, written $e_0 \cong e_1$ just if for any closing context C either both $C[\![e_0]\!]$ and $C[\![e_1]\!]$ are defined or both are undefined.

By definition operational equivalence is a congruence relation on expressions, and not all pairs of expressions are operationally equivalent. In particular, `T` and `Nil` are not operationally equivalent. Also note that returning a value is not the same as being operationally equivalent to a value. This is in contrast to the purely functional case and is due to the presence of *effects*. For example $\mathit{cons}(x, y)$ always returns a value, but is not operationally equivalent to a value.

To define constrained equivalence we need to define strong isomorphism, constraints, and satisfaction. We define $\Gamma; \sigma \models \uparrow e$ to mean that $\mathrm{FV}(\Gamma[\![e_j^{\sigma}]\!]) = \emptyset$ and $\Gamma; e^{\sigma}$ fails to reduce to a value description. Using this notion we say $\Gamma; \sigma \models e_0 \simeq e_1$ (e_0 is stronly isomorphic to e_1 in context $\Gamma; \sigma$) iff $\Gamma; \sigma \models \uparrow e$ for $i \le 2$, or there exist u, Γ', Γ_0, Γ_1 such that $\mathrm{Dom}(\Gamma) \subseteq \mathrm{Dom}(\Gamma')$, $\Gamma'[\![u]\!] \in \mathbb{E}_{\emptyset}$, $\mathrm{Dom}(\Gamma') \cap \mathrm{Dom}(\Gamma_j) = \emptyset$ and $\Gamma; e_j^{\sigma} \overset{*}{\mapsto} (\Gamma_j \cup \Gamma'); u$ for $j < 2$.

The set of constraints $\mathcal{L}$ is a subset of the atomic and negated atomic formulas in the first-order language consisting of equality, the unary function symbols *car* and *cdr*, the unary relation *cell*, and constants from $\mathbb{A}$.

$$\mathcal{L} = (car(\mathbb{U}) = \mathbb{U}) \cup (cdr(\mathbb{U}) = \mathbb{U}) \cup (\mathbb{U} = \mathbb{U}) \cup \neg(\mathbb{U} = \mathbb{U}) \cup (cell(\mathbb{U})) \cup \neg(cell(\mathbb{U}))$$

We let $\varphi, \ldots$ range over $\mathcal{L}$, and $\Sigma, \Sigma_0, \ldots$ range over $\mathbf{P}_\omega(\mathcal{L})$.

A model in our framework is a pair consisting of a memory context and value substitution $\Gamma;\sigma$, such that the range of σ is a subset of $\mathbb{A} \cup \mathrm{Dom}(\Gamma)$. Given a model $\Gamma;\sigma$ and a constraint φ with $\mathrm{FV}(\varphi) \subseteq \mathrm{Dom}(\sigma)$ we say $\Gamma;\sigma$ satisfies φ (written $\Gamma;\sigma \models \varphi$) just if

$$\begin{array}{lcl}
\Gamma;\sigma \models u_0 = u_1 & \Leftrightarrow & u_0^\sigma = u_1^\sigma \\
\Gamma;\sigma \models \neg(u_0 = u_1) & \Leftrightarrow & u_0^\sigma \neq u_1^\sigma \\
\Gamma;\sigma \models car(u_0) = u_1 & \Leftrightarrow & (\exists u \in \mathbb{U})(\Gamma(u_0^\sigma) = [u_1^\sigma, u]) \\
\Gamma;\sigma \models cdr(u_0) = u_1 & \Leftrightarrow & (\exists u \in \mathbb{U})(\Gamma(u_0^\sigma) = [u, u_1^\sigma]) \\
\Gamma;\sigma \models cell(u_0) & \Leftrightarrow & u_0^\sigma \in \mathrm{Dom}(\Gamma) \\
\Gamma;\sigma \models \neg cell(u_0) & \Leftrightarrow & u_0^\sigma \in \mathbb{A}
\end{array}$$

Similarly we define the notion of a model satisfying a set of constraints Σ, (written $\Gamma;\sigma \models \Sigma$). Furthermore we write $\Sigma \models \varphi$ to mean that for every $\Gamma;\sigma$ with $\mathrm{FV}(\varphi) \subseteq \mathrm{Dom}(\sigma) \cup \mathrm{FV}(\Sigma)$ we have that $\Gamma;\sigma \models \Sigma$ implies that $\Gamma, \sigma \models \varphi$. We shall be somewhat free-wheeling with this relation, since by the methods of [11] we can decide whether or not $\Sigma \models \varphi$ in time quadratic in the size of Σ.

We are now in a position to define the notion of constrained equivalence.

Definition ($\Sigma \models e_0 \simeq e_1$): Two expressions e_0, e_1 are equivalent under constraints Σ (written $\Sigma \models e_0 \simeq e_1$) just if $\Gamma;\sigma \models \Sigma$ implies $\Gamma;\sigma \models e_0 \simeq e_1$ for any $\Gamma;\sigma$ with $\mathrm{FV}(\Sigma) \subseteq \mathrm{Dom}(\sigma)$ and $\mathrm{FV}(\Gamma[\![e_j^\sigma]\!]) = \emptyset$ for $j < 2$.

A simple consequence of the definition is that unconstrained equivalence implies operational equivalence: $\emptyset \models e_0 \simeq e_1$ implies $e_0 \cong e_1$.

3. Constraint Propagation

In [5] we presented a formal system for proving assertions of the form

$$\Sigma \vdash \Phi,$$

where Σ is a finite set of constraints and Φ is an element of the assertion language. The assertion language $\mathbb{L}$ is defined as follows:

$$\mathbb{L} = (\mathbb{E} \simeq \mathbb{E}) \cup (\uparrow \mathbb{E})$$

We let $\Phi, \ldots$ range over $\mathbb{L}$. Rather than repeat all the rules here, we have included them in the first appendix. We should point out that officially we should make $\mathcal{D}$ a parameter of the provability 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 set is the following unfolding rule, (**U**), it corresponds to the (**rec**) rule for single-step reduction.

Unfolding rule (U).

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

where $\texttt{<} f(x_1, \ldots x_n) \leftarrow e \texttt{>}$ is in $\mathcal{D}$ and x_i are chosen fresh.

The main results of [5] are that firstly, the deduction system is sound: $\Sigma \vdash \Phi \Rightarrow \Sigma \models \Phi$, and secondly, the deduction system is complete for Φ not containing recursively defined function symbols: $\Sigma \models \Phi \Rightarrow \Sigma \vdash \Phi$.

To facilitate the discussion we define the notion of an assertion Φ holding in a context C as follows:

$$C[\![\Phi]\!] = \begin{cases} \uparrow C[\![e]\!] & \text{if } \Phi = \uparrow e \\ C[\![e_0]\!] \simeq C[\![e_1]\!] & \text{if } \Phi = e_0 \simeq e_1. \end{cases}$$

The system as described in the appendix has several short comings. One difficulty with constrained equivalence is that it is not a congruence. One cannot, in general, place expressions equivalent under some non-empty set of constraints into an arbitrary program context and preserve equivalence. Informally, we say a context C does not invalidate a set of constraints Σ if the following principle is valid.

$$\text{(CI)} \quad \frac{\Sigma \vdash \Phi}{\Sigma \vdash C[\![\Phi]\!]}$$

There are some simple examples of this phenomena. The most trivial case is when Σ is empty. An easy case is when Σ contains only assertions of the form $atom(x)$, $\neg atom(x)$, $x = y$, or $\neg(x = y)$, and C is any context that does not trap the free variables of Σ. A somewhat harder case is when Σ any constraint set and C is of the form $\mathtt{let}\{x := e\}\varepsilon$ where (under constraint Σ) e has no write effect (evaluation of e will not execute any *setcar*s or *setcdr*s) and x is not free in Σ. To solve this problem, as well as others, we extend the system by developing a constraint propagation logic in which we derive assertions of the form

$$\Sigma_0 \vdash C[\![\Sigma_1]\!].$$

We begin by giving the semantics of constraint propatation.

Definition ($\Sigma_0 \models C[\![\Sigma_1]\!]$)**:** A constraint set Σ_0 propagates across context C to Σ_1 (written $\Sigma_0 \models C[\![\Sigma_1]\!]$) just if $\Gamma; C^\sigma \stackrel{*}{\mapsto} \Gamma'; (R[\![\ ]\!])^{\sigma'}$ implies $\Gamma'; \sigma' \models \Sigma_1$, for every $\Gamma; \sigma$ such that $\mathrm{FV}(\Sigma_0) \subseteq \mathrm{Dom}(\sigma)$ and $\mathrm{FV}(\Gamma[\![C^\sigma[\![\Sigma_1^\sigma]\!]]\!] = \emptyset$. (Here, reduction is extended to contexts by carrying along the the substitutions implicit in the let reduction steps.)

One consequence of semantics of constraint propagation is that the following rule is valid.

Constraint Propagation rule (CP).

$$\frac{\Sigma_0 \vdash C[\![\Sigma_1]\!] \quad \Sigma_1 \vdash \Phi}{\Sigma_0 \vdash C[\![\Phi]\!]}$$

Notice that this rule is a variant of the invalid (**CI**) principle. In particular the necessary side condition to validate (**CI**) is that $\Sigma \vdash C[\![\Sigma]\!]$.

Rather than display all the rules here, we restrict our attention to a small representative sample. The full collection needed for the present work may be found in the second appendix. There are six structural rules for this system, two important ones are the following.

Structural rules (c-S).

(i) $\Sigma \vdash R[\![\Sigma]\!]$

(v) $$\frac{\Sigma_0 \vdash C_1[\![\Sigma_1]\!] \qquad \Sigma_1 \vdash C_1[\![\Sigma_2]\!]}{\Sigma_0 \vdash C_0[\![C_1[\![\Sigma_2]\!]]\!]}$$

The rule concerning lexical binding is the following.

Rule for `let` (c-let).

(i) $$\frac{\Sigma \vdash e \simeq \mathtt{seq}(e,u) \quad \Sigma \vdash \mathtt{seq}(e, C[\![\Sigma_0]\!]\{x := u\})}{\Sigma \vdash \mathtt{let}\{x := e\}C[\![\Sigma_0 \cup \{x = u\}]\!]}$$
where $x \notin \mathrm{FV}(\Sigma \cup \Sigma_0)$, and $x, u \notin \mathrm{Traps}(C[\![\ ]\!])$

An important point to observe is that a context can contain more information than is possible to represent by a set of constraints. In particular one may have that $\Sigma \vdash C[\![\Phi]\!]$, yet there is no consistent set of constaints Σ' such that $\Sigma \vdash C[\![\Sigma']\!]$ and $\Sigma' \vdash \Phi$. For example by the lemma (**cons**) (see the first appendix) we have that there is no consistent Σ such that $\Sigma \vdash x \simeq cons(car(x), cdr(x))$ even though every Σ satisfies

$$\Sigma \vdash \mathtt{let}\{x := cons(u_a, u_d)\}[\![x \simeq cons(car(x), cdr(x))]\!]$$

The equation $x \simeq cons(car(x), cdr(x))$ expresses the fact that the cell x is new, in the sense that it is not accessible from any other [non-garbage] cell. Since this is a key property in the validation of many transformations and since it cannot be expressed purely using constraints, we introduce a pseudo constraint $new(x)$. We call constraint sets possibly containing pseudo constraints *invariants* and let $\widehat{\Sigma}$ range over invariants. The meaning of a pseudo constraint depends on whether it appears in the premiss or conclusion.

Definition (Invariant semantics): Let

$$\widehat{\Sigma} = \Sigma \cup \{new(x), car(x) = x_a, cdr(x) = x_d \mid x \in X\}$$

where X is a finite set of variables that do not occur in Σ and are disjoint from $\{x_a, x_d \mid x \in X\}$. Furthermore put $C_X = \mathtt{let}\{x := cons(x_a, x_d)\}_{x \in X}[\![\ ]\!]$. Then

$$\widehat{\Sigma} \vdash \Phi \Leftrightarrow \Sigma \vdash C_X[\![\Phi]\!]$$

$$\widehat{\Sigma} \vdash C[\![\varphi]\!] \Leftrightarrow \Sigma \vdash C_X[\![C[\![\varphi]\!]]\!]$$

$$\widehat{\Sigma} \vdash C[\![new(y)]\!] \Leftrightarrow \Sigma \vdash C_X[\![C[\![cell(y)]\!]]\!] \quad \text{and} \quad \Sigma \vdash C_X[\![C[\![y \simeq cons(car(y), cdr(y))]\!]]\!]$$

Invariants enjoy many of the properties of simple constraints. For example the transitivity rule is valid.

Lemma (C-trans): If $\widehat{\Sigma}_1 \vdash C_1[\![\widehat{\Sigma}_2]\!]$ and $\widehat{\Sigma}_0 \vdash C_0[\![\widehat{\Sigma}_1]\!]$ then $\widehat{\Sigma}_0 \vdash C_0[\![C_1[\![\widehat{\Sigma}_2]\!]]\!]$.

The key property of *new* is expressed by the lemma (**reuse**).

Lemma (reuse):

$$new(x) \vdash cons(u, v) \simeq setpair(x, u, v)$$

or in standard notation

$$\vdash \mathtt{let}\{x := cons(x_a, x_d)[\![cons(u, v) \simeq setpair(x, u, v)]\!]$$

where x_a, x_d are fresh.

4. Tools for Transformations

We now apply constraint propagation and constrained equivalence to the task of program transformation. In particular we wish to describe general methods for establishing the equivalence of related program definitions. We begin with some notation. In the following we let $\mathbf{x}$, $\mathbf{x}_i$, $\mathbf{y}$ be a sequences of variables, and let $\mathbf{u}$ be a sequence of value expressions. We let $\mathbf{f}$ be a sequence of function symbols of length $n+1$ and let f_i denote the i-th symbol of $\mathbf{f}$. We let $\mathcal{D}_\mathbf{f}$ be a system of definitions $[f_i(\mathbf{x}_i) \leftarrow F_i \mid i \leq n]$, possibly referring to previously defined functions. The expression F_i is called the defining expression of f_i in $\mathcal{D}_\mathbf{f}$. The central idea in this section is to formalize the notion of propagating a set of constraints to all important points in an expression, which in this case consist in all occurrences of recursive function calls.

Definition (System of invariants): A system of invariants for $\mathbf{f}$ is a sequence of pairs of invariants $(\widehat{\Sigma}_i^{\mathrm{in}}, \widehat{\Sigma}_i^{\mathrm{out}})$ for $i \leq n$. We let I range over systems of invariants.

The $\widehat{\Sigma}_i^{\mathrm{in}}$ are to be thought of as pre-conditions for f_i, while the $\widehat{\Sigma}_i^{\mathrm{out}}$ are post-conditions. These invariants are used to propagate constraints across any calls to these functions. To make this precise we make the following definition.

Definition (I, f-propagation): Let I be a system of invariants for $\mathbf{f}$, and let $\widehat{\Sigma}, \widehat{\Sigma}'$ be any invariants. We write $\widehat{\Sigma} \vdash_{I,\mathbf{f}} \mathtt{let}\{z := e\}[\![\widehat{\Sigma}']\!]$ to say that $\widehat{\Sigma}$ $I, \mathbf{f}$-propagates across $\mathtt{let}\{z := e\}$ to $\widehat{\Sigma}'$ (taking $\mathbf{x}_i$ to be the formal parameters of f_i for $i \leq n$). This holds just if one of the following:

(i) $\widehat{\Sigma} \vdash \mathtt{let}\{z := e\}[\![\widehat{\Sigma}']\!]$ (without using $\mathcal{D}_\mathbf{f}$);

(ii) $e = f_i(\mathbf{u})$, $\widehat{\Sigma} \vdash (\widehat{\Sigma}_i^{\mathrm{in}})^\sigma$ and $(\widehat{\Sigma}_i^{\mathrm{out}})^\sigma \vdash \widehat{\Sigma}'$, where $\sigma = \{\mathbf{x}_i := \mathbf{u}\}$;

(iii) $e = \mathtt{let}\{z_0 := e_0\}e_1$ and there is some $\widehat{\Sigma}_0$ such that

$$\widehat{\Sigma} \vdash_{I,\mathbf{f}} \mathtt{let}\{z := e_0\}[\![\widehat{\Sigma}_0\{z_0 := z\}]\!]$$

$$\widehat{\Sigma}_0 \vdash_{I,\mathbf{f}} \mathtt{let}\{z := e_1\}[\![\widehat{\Sigma}']\!]$$

(iv) $e = \mathtt{if}(e_0, e_1, e_2)$ and there is some $\widehat{\Sigma}_1, \widehat{\Sigma}_2$ such that

$$\widehat{\Sigma} \vdash \mathtt{if}(e_0, [\![\widehat{\Sigma}_1]\!], [\![\widehat{\Sigma}_2]\!])$$
$$\widehat{\Sigma}_1 \vdash_{I,\mathbf{f}} \mathtt{let}\{z := e_1\}[\![\widehat{\Sigma}']\!]$$
$$\widehat{\Sigma}_2 \vdash_{I,\mathbf{f}} \mathtt{let}\{z := e_2\}[\![\widehat{\Sigma}']\!]$$

(Note that if e_0 contains elements from $\mathbf{f}$ one may need to introduce some lets in order prove propagation.)

Definition (I-inductive): Let I be a system of invariants for $\mathbf{f}$. We say that $\mathcal{D}_{\mathbf{f}}$ is I-inductive if $\widehat{\Sigma}_i^{\mathtt{in}} \vdash_{I,\mathbf{f}} \mathtt{let}\{z := F_i\}[\![\widehat{\Sigma}_i^{\mathtt{out}}]\!]$ for $i \leq n$.

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

$$\widehat{\Sigma}_i^{\mathtt{in}} \vdash \mathtt{let}\{z := f_i(\mathbf{x}_i)\}[\![\widehat{\Sigma}_i^{\mathtt{out}}]\!]$$

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

Proof (si): The validity of this principle is established by a simple computation induction showing the more general fact that if $\mathcal{D}_{\mathbf{f}}$ is I-inductive then $\widehat{\Sigma} \vdash_{I,\mathbf{f}} \mathtt{let}\{z := e\}[\![\widehat{\Sigma}']\!]$ implies $\widehat{\Sigma} \models_{\mathcal{D}_{\mathbf{f}}} \mathtt{let}\{z := e\}[\![\widehat{\Sigma}']\!]$. $\square_{\mathbf{si}}$

Corollary (subgoal induction): To prove $\widehat{\Sigma} \vdash \mathtt{let}\{z := f_0(\mathbf{x}_0)\}[\![\widehat{\Sigma}']\!]$ it suffices to find a system of invariants I for $\mathbf{f}$, such that $\widehat{\Sigma} \vdash \widehat{\Sigma}_0^{\mathtt{in}}$, $\widehat{\Sigma}_0^{\mathtt{out}} \vdash \widehat{\Sigma}'$, and $\mathcal{D}_{\mathbf{f}}$ is I-inductive.

Theorem (recursion induction – simple version): Let $\mathbf{g}$ be a sequence of function symbols of length $n+1$ and let $\mathcal{D}_{\mathbf{g}}$ be a system of definitions $[g_i(\mathbf{x}_i) \leftarrow G_i \mid i \leq n]$. To prove

$$\widehat{\Sigma}_i \vdash f_i(\mathbf{x}_i) \simeq g_i(\mathbf{x}_i)$$

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) $\widehat{\Sigma}_i \vdash \widehat{\Sigma}_i^{\mathtt{in}}$;

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

(iii) $\widehat{\Sigma}_i^{\mathtt{in}} \vdash g_i(\mathbf{x}_i) \simeq F_i\{\mathbf{f} := \mathbf{g}\}$, using $\mathcal{D}_{\mathbf{g}}$; and

(iv) $\widehat{\Sigma}_i^{\mathtt{in}} \vdash f_i(\mathbf{x}_i) \simeq G_i\{\mathbf{g} := \mathbf{f}\}$, using $\mathcal{D}_{\mathbf{f}}$.

Note that the condition (iii) establishes that $f \sqsubseteq g$, while (iv) establishes $g \sqsubseteq f$. As is the case of (**subgoal induction**) the validity of recursion induction is established by simple computation induction. This theorem or principle can be generalized in many ways, one way is to allow for the functions being compared to have different argument lists. This is done in the following version.

Theorem (recursion induction – elaborated): Let $\mathbf{g}$ be a sequence of function symbols of length $n+1$ and let $\mathcal{D}_{\mathbf{g}}$ be a system of definitions $[g_i(\mathbf{y}_i) \leftarrow G_i \mid i \leq n]$ (with the argument lists for for $\mathbf{g}$ possibly being different than those for $\mathbf{f}$). To prove

$$\widehat{\Sigma}_i \vdash f_i(\mathbf{x}_i) \simeq g_i(\mathbf{y}_i)$$

for $i \leq n$, it suffices to find a system of invariants $I = [(\widehat{\Sigma}_i^{\mathrm{in}}, \widehat{\Sigma}_i^{\mathrm{out}}), i \leq n]$ and expressions $e_i^{\mathbf{f}}$, $e_i^{\mathbf{g}}$ such that for $i \leq n$

(i) $\widehat{\Sigma}_i \vdash \widehat{\Sigma}_i^{\mathrm{in}}$

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

(iii) $\widehat{\Sigma}_i^{\mathrm{in}} \vdash g_i(\mathbf{y}_i) \simeq e_i^{\mathbf{g}} \simeq F_i\{f_i := \lambda \mathbf{x}_i.e_i^{\mathbf{g}}, i \leq n\}$ using $\mathcal{D}_{\mathbf{g}}$,

(iv) $\widehat{\Sigma}_i^{\mathrm{in}} \vdash f_i(\mathbf{x}_i) \simeq e_i^{\mathbf{f}} \simeq G_i\{g_i := \lambda \mathbf{y}_i.e_i^{\mathbf{f}}, i \leq n\}$ using $\mathcal{D}_{\mathbf{f}}$.

Corollary (Useless variable introduction/elimination): Let $\mathcal{D}_{\mathbf{g}}$ be obtained from $\mathcal{D}_{\mathbf{f}}$ by adding an additional argument to each function symbol, and passing that argument unchanged in each recursive call. Then $f_i(\mathbf{x}_i) \simeq g_i(\mathbf{x}_i, y)$ for $i < n$.

Proof (useless): The proof is a simple application of the elaborated recursion induction theorem. Propogation is trivial since there are no constraints. Hence it suffices to show that $f \sqsubseteq g$ and $g \sqsubseteq f$, i.e. to establish conditions (iii) and (iv) taking $e_i^{\mathbf{f}} = f_i(\mathbf{x}_i)$ and $e_i^{\mathbf{g}} = g_i(\mathbf{x}_i, y)$. This is trivial, since carrying out the substitutions and unfolding the definitions yields identical pairs of expressions. $\square_{\mathbf{useless}}$

Theorem (Peephole rule): Let $\mathcal{D}_{\mathbf{g}}$ be obtained from $\mathcal{D}_{\mathbf{f}}$ by replacing some $F_i = C[\![e_0]\!]$ by $G_i = C[\![e_1]\!]\{\mathbf{f} := \mathbf{g}\}$. Suppose that $\widehat{\Sigma} \vdash C[\![\widehat{\Sigma}_0]\!]$ and $\widehat{\Sigma}_0 \vdash e_0 \simeq e_1$ uniformly in $\mathbf{f}$, i.e. without using the unfolding rule for any f defined in $\mathcal{D}_{\mathbf{f}}$. Then to prove

$$\widehat{\Sigma} \vdash f_i(\mathbf{x}_i) \simeq g_i(\mathbf{x}_i)$$

for $i \leq n$, it suffices to find a system of invariants $I = [(\widehat{\Sigma}_i^{\mathrm{in}}, \widehat{\Sigma}_i^{\mathrm{out}}), i \leq n]$ such that

- $\widehat{\Sigma} = \widehat{\Sigma}_i^{\mathrm{in}}$, and
- $\mathcal{D}_{\mathbf{f}}$ is I-inductive.

We have chosen a simple instances 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.

5. Transforming the Boyer Benchmark

The source of this example is a Common Lisp version of the well known Boyer benchmark (cf. [1]) that was used as the starting point for developing a parallel version [4]. The heart of the Boyer benchmark program is a rewrite rule based simplifier that repeatedly tries 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.

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.

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 validity of each of the transformation step. 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 Lisp 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.

$$\mathit{term\text{-}mk}(o,a) \leftarrow \mathit{cons}(o,a)$$

$$\mathit{term\text{-}op}(t) \leftarrow \mathit{car}(t)$$

$$\mathit{term\text{-}args}(t) \leftarrow \mathit{cdr}(t)$$

A substitution is a finite map from variables to terms. Substitutions are represented as lists whose entries are pairs consisting of an atom and a term, i.e. as Lisp alists. Maps are extended using $\mathit{acons}(x,y,s)$ which abbreviates $\mathit{cons}(\mathit{cons}(x,y),s)$. A substitution is applied to an atom using *assoc* which returns the (first) pair whose *car* is *eq* to the atom, or `Nil` if no such pair exists. A substitution 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).

5.1.1. The specifiying 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):

$$
\begin{array}{l}
\mathit{match}(t_1, t_2, s) \leftarrow \\
\quad \mathbf{cond}([\mathit{atom}(t_2) \Rightarrow \mathtt{let}\{b := \mathit{assoc}(t_2, s)\} \\
\qquad\qquad\qquad\qquad \mathtt{if}(b, \\
\qquad\qquad\qquad\qquad\quad \mathtt{if}(\mathit{term\text{-}eq}(t_1, \mathit{cdr}(b)), \mathit{cons}(\mathtt{T}, s), \mathit{cons}(\mathtt{Nil}, \mathtt{Nil})), \\
\qquad\qquad\qquad\qquad\quad \mathit{cons}(\mathtt{T}, \mathit{acons}(t_2, t_1, s)))] \\
\qquad\quad [\mathit{atom}(t_1) \Rightarrow \mathit{cons}(\mathtt{Nil}, \mathtt{Nil})] \\
\qquad\quad [\mathit{eq}(\mathit{term\text{-}op}(t_1), \mathit{term\text{-}op}(t_2)) \Rightarrow \mathit{match\text{-}args}(\mathit{term\text{-}args}(t_1), \mathit{term\text{-}args}(t_2), s)] \\
\qquad\quad [\mathtt{T} \Rightarrow \mathit{cons}(\mathtt{Nil}, \mathtt{Nil})])
\end{array}
$$

$$
\begin{array}{l}
\mathit{match\text{-}args}(a_1, a_2, s) \leftarrow \mathtt{if}(\mathbf{and}(\mathit{null}(a_1), \mathit{null}(a_2)), \\
\qquad\qquad\qquad\qquad\qquad \mathit{cons}(\mathtt{T}, s), \\
\qquad\qquad\qquad\qquad\qquad \mathtt{if}(\mathtt{or}(\mathit{null}(a_1), \mathit{null}(a_2)), \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{cons}(\mathtt{Nil}, \mathtt{Nil}), \\
\qquad\qquad\qquad\qquad\qquad\quad \mathtt{let}\{\mathit{fs} := \mathit{match}(\mathit{car}(a_1), \mathit{car}(a_2), s)\} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathtt{if}(\mathit{car}(\mathit{fs}), \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{match\text{-}args}(\mathit{cdr}(a_1), \mathit{cdr}(a_2), \mathit{cdr}(\mathit{fs})), \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{cons}(\mathtt{Nil}, \mathtt{Nil}))))
\end{array}
$$

5.1.2. 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). The transformation from *match* to *match!* proceeds as follows. We introduce a new variable *jnk*, and under the hypothesis that *jnk* denotes a new cell, replace the outer *cons*es constructing the result by assignment to the new cell. We then verify that now the result produced is the new cell passed as a parameter, and hence *jnk* and *fs* in *match-args* can be identified. Now we verify that in addition to $\mathit{new}(\mathit{jnk})$, the invariant $\mathit{cdr}(\mathit{jnk}) = s$ propogates to all points where s is used, hence we may eliminate s as a parameter and replace references to s by $\mathit{cdr}(\mathit{jnk})$. This completes the transformation. The resulting definition of *match!* is the following.

Definition (match!):

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

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

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 only the explicitly introduced constraints. From the transformation we have the following theorem.

Theorem (match):

$$\{\mathit{new}(\mathit{fs}), \mathit{car}(\mathit{fs}) = s_a, \mathit{cdr}(\mathit{fs}) = s\} \vdash \mathit{match}(t_1, t_2, s) \simeq \mathit{match!}(t_1, t_2, \mathit{fs})$$

or in contextual notation

$$\emptyset \vdash \mathtt{let}\{\mathit{fs} := \mathit{cons}(s_a, s)\}[\![\mathit{match}(t_1, t_2, s) \simeq \mathit{match!}(t_1, t_2, \mathit{fs})]\!]$$

5.1.3. Verifying the transformation

Now we outline the formal validation of the transformation steps. The first step transforms *match* to $\mathit{match}^0(t_1, t_2, s, \mathit{jnk})$ by introducing a new parameter *jnk*. By (**useless**), the introduction of *jnk* as an additional parameter produces an equivalent set of definitions.

Theorem (match.0):

$$\emptyset \vdash \mathit{match}^0(t_1, t_2, s, \mathit{jnk}) \simeq \mathit{match}(t_1, t_2, s)$$

$$\emptyset \vdash \mathit{match\text{-}args}^0(a_1, a_2, s, \mathit{jnk}) \simeq \mathit{match\text{-}args}(a_1, a_2, s)$$

The next step constrains applications of match^0 to contexts satisfying $\mathit{new}(\mathit{jnk})$, and transforms match^0 to match^1 by replacing return expressions $\mathit{cons}(u, v)$ by $\mathit{setpair}(\mathit{jnk}, u, v)$.

Definition (match.1):

$$
\begin{array}{l}
\mathit{match}^1(t_1, t_2, s, \mathit{jnk}) \leftarrow \\
\quad \mathtt{cond}([\mathit{atom}(t_2) \Rightarrow \mathtt{let}\{b := \mathit{assoc}(t_2, s))\} \\
\qquad\qquad\qquad \mathtt{if}(b, \\
\qquad\qquad\qquad\quad \mathtt{if}(\mathit{term\text{-}eq}(t_1, \mathit{cdr}(b)), \\
\qquad\qquad\qquad\qquad \mathit{setpair}(\mathit{jnk}, \mathtt{T}, s), \\
\qquad\qquad\qquad\qquad \mathit{setpair}(\mathit{jnk}, \mathtt{Nil}, \mathtt{Nil})), \\
\qquad\qquad\qquad\quad \mathit{setpair}(\mathit{jnk}, \mathtt{T}, \mathit{acons}(t_2, t_1, s)))] \\
\qquad [\mathit{atom}(t_1) \Rightarrow \mathit{setpair}(\mathit{jnk}, \mathtt{Nil}, \mathtt{Nil})] \\
\qquad [\mathit{eq}(\mathit{term\text{-}op}(t_1), \mathit{term\text{-}op}(t_2)) \\
\qquad\quad \Rightarrow \mathit{match\text{-}args}^1(\mathit{term\text{-}args}(t_1), \mathit{term\text{-}args}(t_2), s, \mathit{jnk})] \\
\qquad [\mathtt{T} \Rightarrow \mathit{setpair}(\mathit{jnk}, \mathtt{Nil}, \mathtt{Nil})])
\end{array}
$$

$$
\begin{array}{l}
\mathit{match\text{-}args}^1(a_1, a_2, s, \mathit{jnk}) \leftarrow \mathtt{if}(\mathtt{and}(\mathit{null}(a_1), \mathit{null}(a_2)), \\
\qquad\qquad\qquad \mathtt{seq}(\mathit{setpair}(\mathit{jnk}, \mathtt{T}, s), \mathit{jnk}), \\
\qquad\qquad\qquad \mathtt{if}(\mathtt{or}(\mathit{null}(a_1), \mathit{null}(a_2)), \\
\qquad\qquad\qquad\quad \mathtt{seq}(\mathit{setpair}(\mathit{jnk}, \mathtt{Nil}, \mathtt{Nil}), \mathit{jnk}), \\
\qquad\qquad\qquad\quad \mathtt{let}\{\mathit{fs} := \mathit{match}^1(\mathit{car}(a_1), \mathit{car}(a_2), s, \mathit{jnk})\} \\
\qquad\qquad\qquad\qquad \mathtt{if}(\mathit{car}(\mathit{fs}), \\
\qquad\qquad\qquad\qquad\quad \mathit{match\text{-}args}^1(\mathit{cdr}(a_1), \mathit{cdr}(a_2), \mathit{cdr}(\mathit{fs}), \mathit{jnk}), \\
\qquad\qquad\qquad\qquad\quad \mathtt{seq}(\mathit{setpair}(\mathit{jnk}, \mathtt{Nil}, \mathtt{Nil}), \mathit{jnk}))))
\end{array}
$$

The validity of the $0 \to 1$ transformation follows from (**peephole**) and (**reuse**) using the fact that $\mathit{new}(\mathit{jnk})$ is preserved by match^0 and in fact propagates to every point in the defining expressions. These facts are formalized in the following easy lemmas. Let I^0 be the system of invariants with input and output conditions $\{\mathit{new}(\mathit{jnk})\}$. Thus in I^0 we have

$$\widehat{\Sigma}^{\mathrm{in}}_{m} = \widehat{\Sigma}^{\mathrm{out}}_{m} = \widehat{\Sigma}^{\mathrm{in}}_{ma} = \widehat{\Sigma}^{\mathrm{out}}_{ma} = \{\mathit{new}(\mathit{jnk})\}.$$

Lemma (I^0 propagates): The definition of match^0 is I^0-inductive.

Corollary (I^0 propagates):

$$\widehat{\Sigma}^{\mathrm{in}}_{m,0} \vdash \mathtt{let}\{x := \mathit{match}^0(t_1, t_2, s, \mathit{jnk})\}[\![\mathit{new}(\mathit{jnk})]\!]$$

$$\widehat{\Sigma}^{\mathrm{in}}_{ma,0} \vdash \mathtt{let}\{x := \mathit{match\text{-}args}^0(a_1, a_2, s, \mathit{jnk})\}[\![\mathit{new}(\mathit{jnk})]\!]$$

Lemma (new.jnk): For every context, C, in the definintion of match^0, and $\mathit{match\text{-}args}^0$ respectively.

$$\widehat{\Sigma}^{\mathrm{in}}_{m,0} \vdash C[\![\mathit{new}(\mathit{jnk})]\!]$$

$$\widehat{\Sigma}^{\mathrm{in}}_{ma,0} \vdash C[\![\mathit{new}(\mathit{jnk})]\!]$$

Thus we have the following theorem.

Theorem (match:0-1):

$$\widehat{\Sigma}^{\mathrm{in}}_{m,0} \vdash match^0(t_1, t_2, s, jnk) \simeq match^1(t_1, t_2, s, jnk)$$

In the next step $match^1$ is transformed into $match^2$ by replacing *fs* by *jnk* in $match\text{-}args^1$. Let I^1 be the system of invariants obtained from I^0 by adding $z = jnk$ to the output conditions. The validity of the $1 \to 2$ transformation is due to (**peephole**) using the fact (**eq.jnk**) that the cell returned by $match^1(t_1, t_2, s, jnk)$ is *jnk*. The proof of this lemma also shows that I^1 propagates across $match^1$ thus validating the application of the peephole rule.

Lemma (eq.jnk):

$$\widehat{\Sigma}^{\mathrm{in}}_{m,1} \vdash \mathtt{let}\{z := match^1(t_1, t_2, s, jnk)\}[\![z = jnk]\!]$$

$$\widehat{\Sigma}^{\mathrm{in}}_{ma,1} \vdash \mathtt{let}\{z := match\text{-}args^1(a_1, ta2, s, jnk)\}[\![z = jnk]\!]$$

Proof (eq.junk): We show that the definition of $match^1$ is I^1-inductive. The lemma then follows by subgoal induction. Note that in I^1 we have

$$\widehat{\Sigma}^{\mathrm{in}}_{m,1} = \{new(jnk)\}$$

$$\widehat{\Sigma}^{\mathrm{out}}_{m,1} = \{new(jnk), z = jnk\}$$

$$\widehat{\Sigma}^{\mathrm{in}}_{ma,1} = \{new(jnk)\}$$

$$\widehat{\Sigma}^{\mathrm{out}}_{ma,1} = \{new(jnk), z = jnk\}$$

To show that $\widehat{\Sigma}^{\mathrm{in}}_{m,1}$ propagates across the defining expression of $match^1$ to $\widehat{\Sigma}^{\mathrm{out}}_{m,1}$ we define the following additional invariants: $\widehat{\Sigma}_1, \widehat{\Sigma}_2$ for the branches of the $atom(t_2)$ test and $\widehat{\Sigma}_{21}, \widehat{\Sigma}_{22}$ for the branches of the $atom(t_1)$ test. We also use $\widehat{\Sigma}_{22}$ for both branches of the *eq* test as the information in this test is not needed.

$$\widehat{\Sigma}_1 = \widehat{\Sigma}^{\mathrm{in}}_{m,1} \cup \{atom(t_2)\}$$

$$\widehat{\Sigma}_2 = \widehat{\Sigma}^{\mathrm{in}}_{m,1} \cup \{\neg atom(t_2)\}$$

$$\widehat{\Sigma}_{21} = \widehat{\Sigma}^{\mathrm{in}}_2 \cup \{atom(t_1)\}$$

$$\widehat{\Sigma}_{22} = \widehat{\Sigma}^{\mathrm{in}}_2 \cup \{\neg atom(t_1)\}$$

Now we need only verify the following assertions. (Which is easily done noting the fact that *setpair* returns its first argument, i.e. $cell(x) \vdash \mathtt{let}\{z := setpair(x, u, v)\}[\![z = x]\!]$).

(0) $\widehat{\Sigma}^{\mathrm{in}}_{m,1} \vdash \mathtt{if}(atom(t_2), [\![\widehat{\Sigma}_1]\!], [\![\widehat{\Sigma}_2]\!])$

(1) $\widehat{\Sigma}_1 \vdash \mathtt{let}\{z := e_0\}[\![\widehat{\Sigma}^{\mathrm{out}}_{m,1}]\!]$ where

$$e_0 = \mathtt{let}\{b := \mathit{assoc}(t_2, s)\} \\ \mathtt{if}(b, \\ \mathtt{if}(\mathit{term\text{-}eq}(t_1, \mathit{cdr}(b)), \mathit{setpair}(\mathit{jnk}, \mathtt{T}, s), \mathit{setpair}(\mathit{jnk}, \mathtt{Nil}, \mathtt{Nil})), \\ \mathit{setpair}(\mathit{jnk}, \mathtt{T}, \mathit{acons}(t_2, t_1, s)))]$$

(2) $\widehat{\Sigma}_2 \vdash \mathtt{if}(\mathit{atom}(t_1), [\![\widehat{\Sigma}_{21}]\!], [\![\widehat{\Sigma}_{22}]\!])$

(2.1) $\widehat{\Sigma}_{21} \vdash \mathtt{let}\{z := \mathit{setpair}(\mathit{jnk}, \mathtt{Nil}, \mathtt{Nil})\}[\![\widehat{\Sigma}^{\mathrm{out}}_{m,1}]\!]$

(2.2) $\widehat{\Sigma}_{22} \vdash \mathtt{if}(\mathit{eq}(\mathit{term\text{-}op}(t_1), \mathit{term\text{-}op}(t_2)), [\![\widehat{\Sigma}_{22}]\!], [\![\widehat{\Sigma}_{22}]\!])$

(2.2.1) $\widehat{\Sigma}_{22} \vdash \mathtt{let}\{a_1 := \mathit{term\text{-}args}(t_1)\}\mathtt{let}\{a_2 := \mathit{term\text{-}args}(t_2)\}[\![\widehat{\Sigma}^{\mathrm{in}}_{ma,1}]\!]$ and $\widehat{\Sigma}^{\mathrm{out}}_{ma,1} \vdash \widehat{\Sigma}^{\mathrm{out}}_{m,1}$

(2.2.2) $\widehat{\Sigma}_{22} \vdash \mathtt{let}\{z := \mathit{setpair}(\mathit{jnk}, \mathtt{Nil}, \mathtt{Nil})\}[\![\widehat{\Sigma}^{\mathrm{out}}_{m,1}]\!]$

$\Box_{\mathbf{m}}$

To show that $\widehat{\Sigma}^{\mathrm{in}}_{ma,1}$ propagates across the defining expression of *match-args*[1] to $\widehat{\Sigma}^{\mathrm{out}}_{ma,1}$ we define the following additional invariants: $\widehat{\Sigma}_1, \widehat{\Sigma}_2$ for the branches of the **and** test and $\widehat{\Sigma}_{21}, \widehat{\Sigma}_{22}$ for the branches of the **or** test. We also use $\widehat{\Sigma}_{22}$ for both branches of the f test as the information in this test is not needed.

$$\widehat{\Sigma}_1 = \widehat{\Sigma}^{\mathrm{in}}_{m,1} \cup \{a_1 = \mathtt{Nil}, a_2 = \mathtt{Nil}\}$$

$$\widehat{\Sigma}_2 = \widehat{\Sigma}^{\mathrm{in}}_{m,1}$$

$$\widehat{\Sigma}_{21} = \widehat{\Sigma}_2$$

$$\widehat{\Sigma}_{22} = \widehat{\Sigma}_2 \cup \{a_1 \neq \mathtt{Nil}, a_2 \neq \mathtt{Nil}\}$$

Now we need only verify the following assertions, which again is straight forward.

(0) $\widehat{\Sigma}^{\mathrm{in}}_{ma,1} \vdash \mathtt{if}(\mathbf{and}(\mathit{null}(a_1), \mathit{null}(a_2)), [\![\widehat{\Sigma}_1]\!], [\![\widehat{\Sigma}_2]\!])$

(1) $\widehat{\Sigma}_1 \vdash \mathtt{let}\{z := \mathit{setpair}(\mathit{jnk}, \mathtt{T}, s)\}[\![\widehat{\Sigma}^{\mathrm{out}}_{ma,1}]\!]$

(2) $\widehat{\Sigma}_2 \vdash \mathtt{if}(\mathbf{or}(\mathit{null}(a_1), \mathit{null}(a_2)), [\![\widehat{\Sigma}_{21}]\!], [\![\widehat{\Sigma}_{22}]\!])$

(2.1) $\widehat{\Sigma}_{21} \vdash \mathtt{let}\{z := \mathit{setpair}(\mathit{jnk}, \mathtt{T}, s)\}[\![\widehat{\Sigma}^{\mathrm{out}}_{ma,1}]\!]$

(2.2) $\widehat{\Sigma}_{22} \vdash \mathtt{let}\{t_1 := \mathit{car}(a_1)\}\mathtt{let}\{t_2 := \mathit{car}(a_2)\}[\![\widehat{\Sigma}^{\mathrm{in}}_{m}]\!]$

(2.2.0) $\widehat{\Sigma}_{22} \cup \widehat{\Sigma}^{\mathrm{out}}_{m,1}\{z := \mathit{fs}\} \vdash$

$$\mathtt{let}\{a_1 := \mathit{cdr}(a_1)\}\mathtt{let}\{a_2 := \mathit{cdr}(a_2)\}\mathtt{let}\{s := \mathit{cdr}(\mathit{fs})\}\mathtt{if}(f, [\![\widehat{\Sigma}^{\mathrm{in}}_{ma,1}]\!], [\![\widehat{\Sigma}^{\mathrm{in}}_{ma,1}]\!])$$

(2.2.2) $\widehat{\Sigma}^{\mathrm{in}}_{ma,1} \vdash \mathtt{let}\{z := \mathit{setpair}(\mathit{jnk}, \mathtt{Nil}, \mathtt{Nil})\}[\![\widehat{\Sigma}^{\mathrm{out}}_{ma,1}]\!]$

$\Box_{\mathbf{ma}}$ $\Box_{\mathbf{match.new}}$

In the next step we add the constraint $\mathit{cdr}(\mathit{fs}) = s$ and transform match^2 to match^3 by replacing references to s by $\mathit{cdr}(\mathit{fs})$. This is valid by (**peephole**) and (**FS**) below.

Lemma (FS): For each recursive calling-context, C, in the $match^2$ definition.

$$\{new(fs), cdr(fs) = s\} \vdash C[\![cdr(fs) = s]\!]$$

$$\{new(fs), cdr(fs) = s\} \vdash C[\![cdr(fs) = s]\!]$$

In the final step we remove the useless argument, s, obtaining the *match!* definition. This is again valid by (**useless**). Some additional consequences of our transformation are the following.

Corollary (match):

$$\emptyset \vdash \mathtt{let}\{z := match(t_1, t_2, s)\}[\![new(z)]\!]$$

$$\{new(fs), cdr(fs) = s\} \vdash match(t_1, t_2, s) \simeq \mathtt{seq}(setpair(fs, \mathtt{T}, s), match!(t_1, t_2, fs), fs)$$

5.2. Transforming the rewriter

We extend the notation for matching. A lemma has a lefthand side and a righthand side, each of which is a term.

$$lemma\text{-}lhs(x) \leftarrow car(x)$$

$$lemma\text{-}rhs(x) \leftarrow cdr(x)$$

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

5.2.1. 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):

$$\begin{array}{l}
rewrite(t, l) \leftarrow \mathtt{if}(atom(t), \\
\qquad\qquad t, \\
\qquad\qquad \mathtt{let}\{t' := term\text{-}mk(term\text{-}op(t), rewrite\text{-}args(term\text{-}args(t), l))\} \\
\qquad\qquad\quad rewrite\text{-}top(t', l, l))
\end{array}$$

$$\begin{array}{l}
rewrite\text{-}args(a, l) \leftarrow \mathtt{if}(null(a), \\
\qquad\qquad \mathtt{Nil}, \\
\qquad\qquad cons(rewrite(car(a), l), rewrite\text{-}args(cdr(a), l))
\end{array}$$

$$\begin{array}{l}
rewrite\text{-}top(t, cl, l) \leftarrow \\
\quad \mathtt{if}(null(cl), \\
\quad\quad t, \\
\quad\quad \mathtt{let}\{fs := match(t, lemma\text{-}lhs(car(cl)), \mathtt{Nil})\} \\
\quad\quad\quad \mathtt{if}(car(fs), \\
\quad\quad\quad\quad rewrite(app\text{-}sbst(cdr(fs), lemma\text{-}rhs(car(cl))), l), \\
\quad\quad\quad\quad rewrite\text{-}top(t, cdr(cl), l)))
\end{array}$$

5.2.2. Transformation steps

We transform *rewrite* into *rewrite!* as follows. First introduce an additional parameter *jnk* using (**useless**). Then we verify that $new(jnk)$ propogates as for *match*. Using this and properties of *match* and *match!* we apply the peephole rule to replace the call to *match* by initialization of *jnk* and a call to *match!*. Then using the second corollary to (**match**) we can replace *fs* by *jnk* following the binding of *fs* [and then rename *jnk* to *fs*].

Definition (rewrite!):

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

From the validity of the transformation we have the following theorem.

Theorem (rewrite):

(rw) $\{new(\mathit{fs})\} \vdash \mathit{rewrite}(t, l) \simeq \mathit{rewrite!}(t, l, \mathit{fs})$

(rwa) $\{new(\mathit{fs})\} \vdash \mathit{rewrite\text{-}args}(a, l) \simeq \mathit{rewrite\text{-}args!}(a, l, \mathit{fs})$

(rwt) $\{new(\mathit{fs})\} \vdash \mathit{rewrite\text{-}top}(t, \mathit{cl}, l) \simeq \mathit{rewrite\text{-}top!}(t, \mathit{cl}, l, \mathit{fs})$

6. Conclusion

In this paper we have presented rules for systematically transforming programs in 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 constraint propagation. Subgoal and recursion induction are standard methods of proving properties of flow-chart and first-order programs acting on immutable data [8, 10]. We have generalized these methods to programs that act on mutable data structures. The extended constraint notation allows one to express notions such as *exclusive use* which allow one to replace allocation by updating [2].

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. The original application of the 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 *match* and *rewrite* transformations are refinements in the sense that they involve accumulation of information, addition of constraints to the interface (input, output conditions), and making use of the additional information. Thus at each stage, validation of the application of a transformation rule is an incremental process. Constraint propagation proofs required for validation of previous rules can be "reused" and only newly added constraints require any work.

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 [7] 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.

The pseudo constraint *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

This research was partially supported by DARPA contract N00039-84-C-0211 and NSF grant CCR-8718605.

7. References

[1] Richard P. Gabriel. *Performance and Evaluation of Lisp Systems.* Computer Systems Series. MIT Press, Cambridge, Massachusetts, 1985.

[2] 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.

[3] 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.

[4] I. A. Mason, J. D. Pehoushek, C. L. Talcott, and J. S. Weening. Programming in Qlisp. Technical Report to appear, Department of Computer Science, Stanford University, 1990.

[5] 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.

[6] 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*, 1989.

[7] I. A. Mason and C. L. Talcott. A sound and complete axiomatization of operational equivalence between programs with memory. *Theoretical Computer Science*, to appear, 199?

[8] 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.

[9] E. Moggi. Computational lambda-calculus and monads. In *Fourth Annual Symposium on logic in computer science.* IEEE, 1989.

[10] J. H. Morris and B. Wegbreit. Subgoal induction. *Communications of the Association for Computing Machinery*, 20:209–222, 1976.

[11] C. G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. Technical Report STAN-CS-77-647, Department of Computer Science, Stanford University, 1977.

[12] G. Plotkin. Call-by-name, call-by-value and the lambda calculus. *Theoretical Computer Science*, 1, 1975.

8. Appendix A: The Formal System

Definition ($\Sigma \vdash \Phi$): The consequence relation, $\vdash$, is the smallest relation that is closed under the rules given below.

The rules are partitioned into several groups of related rules. Each group of rules is given a label for future reference and members of the group are numbered. For example (**S.i**) refers to the first rule in the group of structural rules (the first group below). A rule has a (possibly empty) set of premisses and a conclusion. In the case that the set of premisses is non-empty the rule is displayed with a horizontal bar separating the premisses from the conclusion. A pair of rules that differ by interchanging premiss and conclusion is presented as a single rule with a double bar.

We begin with the structural rules. (**S.i**) asserts the obvious connection between a constraint φ and its translation into the assertion language $T(\varphi)$, defined by

$$T(\varphi) = \begin{cases} \vartheta(x) \simeq u & \text{if } \varphi \text{ is } \vartheta(x) = u \text{ and } \vartheta \in \{car, cdr\} \\ eq(u_0, u_1) \simeq \mathtt{T} & \text{if } \varphi \text{ is } u_0 = u_1 \\ eq(u_0, u_1) \simeq \mathtt{Nil} & \text{if } \varphi \text{ is } \neg(u_0 = u_1) \\ cell(x) \simeq \mathtt{T} & \text{if } \varphi \text{ is } cell(x) \\ cell(x) \simeq \mathtt{Nil} & \text{if } \varphi \text{ is } \neg cell(x). \end{cases}$$

The second structural rule, (**S.ii**), allows us to replace any set of constraints with a logically equivalent set. It is used to put constraint sets into a form necessary for application of another rule — for example (**set.vii**). It also incorporates trivial facts concerning equality and the nature of the atoms.

Structural rules (S).

(i) $\Sigma \cup \{\varphi\} \vdash T(\varphi)$

(ii) $$\frac{\Sigma \cup \{\varphi\} \vdash \Phi}{\Sigma \vdash \Phi}\ (\text{double bar}) \qquad \text{assuming } \Sigma \models \varphi$$

The left elimination rules, (**L.i, L.ii**), enable one to reason by cases, while (**L.iii**) allows us to name the *car* and *cdr* of a cell.

Left elimination (L).

(i) $$\frac{\Sigma \cup \{cell(x)\} \vdash \Phi \qquad \Sigma \cup \{\neg cell(x)\} \vdash \Phi}{\Sigma \vdash \Phi}$$

(ii) $$\frac{\Sigma \cup \{\neg(u_0 = u_1)\} \vdash \Phi \qquad \Sigma \cup \{u_0 = u_1\} \vdash \Phi}{\Sigma \vdash \Phi}$$

(iii) $$\frac{\Sigma \cup \{\vartheta(x) = z\} \vdash \Phi}{\Sigma \vdash \Phi}$$

assuming that $\Sigma \models cell(x)$ and $z \notin \mathrm{FV}(\Phi) \cup \mathrm{FV}(\Sigma)$

The fact that $\simeq$ is an equivalence relation is expressed by the rules (**E.i, E.ii, E.iii**), while (**eq.i**) asserts that *eq* objects are also strongly isomorphic.

Equivalence rules (E).

(i) $\Sigma \vdash e_0 \simeq e_0$ (ii) $\dfrac{\Sigma \vdash e_0 \simeq e_1 \quad \Sigma \vdash e_1 \simeq e_2}{\Sigma \vdash e_0 \simeq e_2}$ (iii) $\dfrac{\Sigma \vdash e_0 \simeq e_1}{\Sigma \vdash e_1 \simeq e_0}$

Rule concerning *eq* **(eq).**

(i) $\dfrac{\Sigma \vdash eq(x, y) \simeq \mathtt{T}}{\Sigma \vdash x \simeq y}$

The rules (**D**) express our treatment of error situations as a form of failure to return a value, i.e. as a form of divergence. Two divergent expressions are strongly isomorphic, (**D.i**). An expression strongly isomorphic to a divergent expression is itself divergent, (**D.ii**). The memory operations ϑ and $set\vartheta$ diverge when their first argument is not a cell, (**D.iii, D.iv**)

Divergence rules (D).

(i) $\dfrac{\Sigma \vdash \uparrow e_0 \qquad \Sigma \vdash \uparrow e_1}{\Sigma \vdash e_0 \simeq e_1}$ (ii) $\dfrac{\Sigma \vdash \uparrow e_0 \qquad \Sigma \vdash e_0 \simeq e_1}{\Sigma \vdash \uparrow e_1}$

(iii) $\dfrac{\Sigma \vdash cell(x) \simeq \mathtt{Nil}}{\Sigma \vdash \uparrow \vartheta(x)}$ (iv) $\dfrac{\Sigma \vdash cell(x) \simeq \mathtt{Nil}}{\Sigma \vdash \uparrow set\vartheta(x, y)}$

The reduction context rules express the extent to which constrained equivalence is a congruence. Note that all these rules are false for arbitrary contexts. In the case of $\uparrow$ assertions, (**R.i**) determines how this information is propagated.

Reduction context rules (R).

(i) $\dfrac{\Sigma \vdash \Phi}{\Sigma \vdash R[\![\Phi]\!]}$

(ii) $\Sigma \vdash R[\![\mathtt{if}(e_0, e_1, e_2)]\!] \simeq \mathtt{if}(e_0, R[\![e_1]\!], R[\![e_2]\!])$

(iii) $\Sigma \vdash R[\![\mathtt{let}\{x := e_0\}e_1]\!] \simeq \mathtt{let}\{x := e_0\}R[\![e_1]\!] \qquad x \notin \mathrm{FV}(R)$

The first `let` rule expresses the fact that the identity applied to an expression is equivalent to the expression itself. The second rule is simply beta-value reduction restricted to the first-order notion of a value.

Rules concerning `let` (let).

(i) $\Sigma \vdash e \simeq \mathtt{let}\{x := e\}x$ (ii) $\Sigma \vdash e\{x := u\} \simeq \mathtt{let}\{x := u\}e$

There are three rules concerning `if`. The first expresses a simple connection between `if` and `let`. The second expresses the fact that an `if` reduces to its else clause when its test evaluates to `Nil`. The third expresses the fact that an `if` reduces to its else clause when the test evaluates to a non-`Nil` value.

Rules concerning `if` (if).

(i) $\Sigma \vdash \mathtt{seq}(e_0, e_1) \simeq \mathtt{let}\{x := e_0\}e_1$ assuming that $x \notin \mathrm{FV}(e_1)$

recall that $\mathtt{seq}(e_0, e_1)$ abbreviates $\mathtt{if}(e_0, e_1, e_1)$

(ii) $\Sigma \vdash \mathtt{if}(\mathtt{Nil}, e_0, e_1) \simeq e_1$ (iii) $\dfrac{\Sigma \vdash eq(u, \mathtt{Nil}) \simeq \mathtt{Nil}}{\Sigma \vdash \mathtt{if}(u, e_0, e_1) \simeq e_0}$

There are three rules concerning *cons*. The first two assert that the time of allocation is irrelevant to the result of a *cons*. The third rule is essentially an introduction rule for *cons*. It expresses the fact that *cons* allocates a new cell with its arguments as the cells contents.

Rules for *cons* (cons).

(i) $\Sigma \vdash \mathtt{let}\{x_0 := cons(\mathtt{T},\mathtt{T})\}\mathtt{let}\{x_1 := cons(\mathtt{T},\mathtt{T})\}e$

$\simeq \mathtt{let}\{x_1 := cons(\mathtt{T},\mathtt{T})\}\mathtt{let}\{x_0 := cons(\mathtt{T},\mathtt{T})\}e$

(ii) $\Sigma \vdash \mathtt{seq}(e_0, \mathtt{let}\{x := cons(u_0, u_1)\}e_1) \simeq \mathtt{let}\{x := cons(u_0, u_1)\}\mathtt{seq}(e_0, e_1)$

subject to the condition that $x \notin \mathrm{FV}(e_0)$

(iii) $\dfrac{\Sigma \cup \Sigma_x \vdash \Phi}{\Sigma \vdash \mathtt{let}\{x := cons(u_a, u_d)\}[\![\Phi]\!]}$ $x \notin (\mathrm{FV}(\Sigma) \cup \{u_a, u_d\}) = Z$ and

$\Sigma_x = \{cell(x), car(x) = u_a, cdr(x) = u_d, \neg(x = y) \mid y \in Z \cup \mathrm{FV}(\Phi) - \{x\}\}$

There are seven rules concerning the $set\vartheta$ operations. (**set.i**) asserts that two successive $set\vartheta$ operations may be commuted, provided that they are altering distinct cells. (**set.ii**) asserts that successive modifications to the same cell is equivalent to simply carrying out the second modification. (**set.iii**) asserts that $set\vartheta$ returns as its value the modified cell. (**set.iv**) asserts that successive *setcar* and *setcdr* operations may always be commuted. (**set.v, set.vi**) assert that updating a newly allocated cell is equivalent to allocating it with the more recent contents. (**set.vii**) is an introduction and elimination rule for $set\vartheta$. The side condition that x is ϑ-less in Σ is a technicality that needed to ensure, amongst other things, that the consistency of $\Sigma \cup \{\vartheta(x) = u_0\}$ is equivalent to the consistency of $\Sigma \cup \{\vartheta(x) = u_1\}$. It is formally defined as follows, x is ϑ-less in Σ just if $\neg(\exists u \in \mathbb{U})(\Sigma \models \vartheta(x) = u)$ and $(\forall y \in \mathbb{X})((\vartheta(y) = u) \in \Sigma \Rightarrow \Sigma \models \neg(x = y))$. This side condition is discussed more fully in [Mason and Talcott, 1990??].

Rules for *setcar* and *setcdr* (set).

(i) $\dfrac{\Sigma \vdash eq(x_0, x_2) \simeq \mathtt{Nil}}{\Sigma \vdash \mathtt{seq}(set\vartheta(x_0, x_1), set\vartheta(x_2, x_3), e) \simeq \mathtt{seq}(set\vartheta(x_2, x_3), set\vartheta(x_0, x_1), e)}$

(ii) $\Sigma \vdash \mathtt{seq}(set\vartheta(x, y_0), set\vartheta(x, y_1)) \simeq set\vartheta(x, y_1)$

(iii) $\Sigma \vdash \mathtt{seq}(set\vartheta(x, y), x) \simeq set\vartheta(x, y)$

(iv) $\Sigma \vdash \mathtt{seq}(setcdr(x_0, x_1), setcar(x_2, x_3), e) \simeq \mathtt{seq}(setcar(x_2, x_3), setcdr(x_0, x_1), e)$

(v) $\Sigma \vdash setcar(cons(z, y), x) \simeq cons(x, y)$

(vi) $\Sigma \vdash setcdr(cons(x,z),y) \simeq cons(x,y)$

(vii) $$\frac{\Sigma \cup \{\vartheta(x) = u_0\} \vdash \Phi}{\Sigma \cup \{\vartheta(x) = u_1\} \vdash \mathtt{seq}(set\vartheta(x,u_0), [\![\Phi]\!])}$$

assuming $\Sigma \models cell(x)$ and x is ϑ-less in Σ

The garbage collection rule allows for the elimination of garbage — *cons* cells no longer accessible from the program text.

Garbage collection rule (G). If Γ is a context of the form

$$\mathtt{let}\{z_1 := cons(\mathtt{T},\mathtt{T})\}\ldots\mathtt{let}\{z_n := cons(\mathtt{T},\mathtt{T}))\}$$
$$\mathtt{seq}(setcar(z_1,u_1^a), setcdr(z_1,u_1^d), \ldots, setcar(z_n,u_n^a), setcdr(z_n,u_n^d), \varepsilon).$$

and $\{z_1,\ldots,z_n\} \cap \mathrm{FV}(e) = \emptyset$, then

$\Sigma \vdash \Gamma[\![e]\!] \simeq e.$

The unfolding rule (**U**) corresponds to the (**rec**) rule for single-step reduction. It is necessary in order to account for evaluation of recursively defined function symbols.

Unfolding rule (U).

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

where $<f(x_1,\ldots x_n) \leftarrow e>$ is in $\mathcal{D}$ and x_i are chosen fresh.

The inference system presented above is minimal by design in order to simplify the proof of completeness. The choice of rules is not necessarily the best if we want a basis that extends nicely. In particular a number of sound rules that are derivable in that system are no longer derivable when new rules, such as rank induction, are added. Two such rules are (**monotonicity**), (**equality**). For the present we shall add these as official rules of inference.

Monotonicity rule (Mon).

(monotonicity) $$\frac{\Sigma \vdash \Phi}{\Sigma \cup \Sigma' \vdash \Phi}$$

Equality rule (Eq).

(equality) $$\frac{\Sigma \cup \{x = y\} \vdash \Phi}{\Sigma \cup \{x = y\} \vdash \Phi\{x := y\}}$$

As a simple application of the rules we prove the following lemma.

Lemma (cons): The following rule is derivable

$$\frac{\Sigma \vdash x \simeq cons(car(x), cdr(x))}{\Sigma \vdash \mathtt{T} \simeq \mathtt{Nil}}$$

Proof (cons): By choosing x_a, x_d fresh and using (**L.iii**) we may assume that $\{car(x) = x_a, cdr(x) = x_d\} \subset \Sigma$. Now let $(\mathrm{FV}(\Sigma) \cup \{x_a, x_d\}) = Z$ and

$$\Sigma_y = \{cell(y), car(y) = x_a, cdr(y) = x_d, \neg(z = y) \mid z \in Z \cup \mathrm{FV}(\Phi) - \{y\}\}.$$

Now since $x \in \mathrm{FV}(\Sigma)$, we have the following instantiation of (**cons.iii**).

$$\frac{\Sigma \cup \Sigma_y \vdash eq(x, y) \simeq \mathtt{Nil}}{\Sigma \vdash \mathtt{let}\{y := cons(x_a, x_d)\}[\![eq(x, y) \simeq \mathtt{Nil}]\!]}$$

Since the hypothesis is a consequence of (**S.i**) we may conclude that

$$\Sigma \vdash \mathtt{let}\{y := cons(x_a, x_d)\}[\![eq(x, y) \simeq \mathtt{Nil}]\!]$$

Now by (**let.i,R.iii,E,G**) this implies that

$$\Sigma \vdash eq(x, cons(x_a, x_d)) \simeq \mathtt{Nil}$$

Using the hypothesis and (**R.i,E**) we conclude that

$$\Sigma \vdash eq(x, x) \simeq \mathtt{Nil}$$

Hence by (**eq.i**) and the equivalence rules we have shown that $\Sigma \vdash \mathtt{T} \simeq \mathtt{Nil}$ $\square_{\mathbf{cons}}$

9. Appendix B: The Constraint Propagation Rules

The structural rules for this system are the following.

Structural rules (c-S).

(i) $\Sigma \vdash R[\![\Sigma]\!]$

(ii) $\dfrac{\Sigma' \vdash C[\![\Sigma_0]\!]}{\Sigma \vdash C[\![\Sigma_0]\!]}$ assuming $\Sigma \vdash \varphi$ for each $\varphi \in \Sigma'$

(iii) $\dfrac{\Sigma_0' \vdash C[\![\Sigma_1']\!] \quad \Sigma_0' \subseteq \Sigma_0 \quad \Sigma_1' \supseteq \Sigma_1}{\Sigma_0 \vdash C[\![\Sigma_1]\!]}$

(iv) $\dfrac{\Sigma_0 \vdash C[\![\Sigma_1]\!] \quad \Sigma_0' \vdash C[\![\Sigma_1']\!]}{\Sigma_0 \cup \Sigma_0' \vdash C[\![\Sigma_1 \cup \Sigma_1']\!]}$

(v) $\dfrac{\Sigma_0 \vdash C_1[\![\Sigma_1]\!] \qquad \Sigma_1 \vdash C_1[\![\Sigma_2]\!]}{\Sigma_0 \vdash C_0[\![C_1[\![\Sigma_2]\!]]\!]}$

(vi) $\dfrac{\Sigma_0 \vdash e_0 \simeq e_1 \quad \Sigma_0 \vdash \mathtt{let}\{x := e_0\}C[\![\Sigma_1]\!]}{\Sigma_0 \vdash \mathtt{let}\{x := e_1\}C[\![\Sigma_1]\!]}$

Left elimination (c-L).

$$\text{(i)}\quad \frac{\Sigma \cup \{cell(x)\} \vdash C[\![\Sigma']\!] \qquad \Sigma \cup \{\neg cell(x)\} \vdash C[\![\Sigma']\!]}{\Sigma \vdash C[\![\Sigma']\!]}$$

$$\text{(ii)}\quad \frac{\Sigma \cup \{\neg(u_0 = u_1)\} \vdash C[\![\Sigma']\!] \qquad \Sigma \cup \{u_0 = u_1\} \vdash C[\![\Sigma']\!]}{\Sigma \vdash C[\![\Sigma']\!]}$$

$$\text{(iii)}\quad \frac{\Sigma \cup \{\vartheta(x) = z\} \vdash C[\![\Sigma']\!]}{\Sigma \vdash C[\![\Sigma']\!]} \qquad \text{where } \vartheta \in \{car, cdr\}, \Sigma \models cell(x), \text{ and } z \text{ is fresh}$$

Rule for `if` (c-if).

$$\text{(i)}\quad \frac{\Sigma \cup \{\neg(u = \mathtt{Nil})\} \vdash C_1[\![\Sigma_1]\!]}{\Sigma \vdash \mathtt{if}(u, C_1[\![\Sigma_1]\!], e_2)}$$

$$\text{(ii)}\quad \frac{\Sigma \cup \{u = \mathtt{Nil}\} \vdash C_2[\![\Sigma_1]\!]}{\Sigma \vdash \mathtt{if}(u, e_1, C_2[\![\Sigma_2]\!])}$$

Rule for `let` (c-let).

$$\text{(i)}\quad \frac{\Sigma \vdash e \simeq \mathtt{seq}(e, u) \quad \Sigma \vdash \mathtt{seq}(e, C[\![\Sigma_0]\!]\{x := u\})}{\Sigma \vdash \mathtt{let}\{x := e\}C[\![\Sigma_0 \cup \{x = u\}]\!]}$$

where $x \notin \mathrm{FV}(\Sigma \cup \Sigma_0)$, and $x, u \notin \mathrm{Traps}(C[\![\ ]\!])$

Rule for *cons* (c-cons).

(i) $\Sigma \vdash \mathtt{let}\{x := cons(u_a, u_d)\}[\![\Sigma \cup \Delta]\!] \qquad x \notin Z = (\mathrm{FV}(\Sigma) \cup \{u_a, u_d\})$

where $\Delta = \{cell(x), car(x) = u_a, cdr(x) = u_d, \neg(x = y) \mid y \in Z\}$

Rules for *set*ϑ (c-set).

(i) $\Sigma \vdash \mathtt{seq}(set\vartheta(x, u_0), [\![\Sigma]\!])$

where $x \notin \mathrm{FV}(\Sigma)$, $set\vartheta \in \{setcar, setcdr\}$

(ii) $\{cell(x), car(x) = u_1\} \vdash \mathtt{seq}(setcar(x, u_0), [\![\{cell(x), car(x) = u_0\}]\!])$

(iii) $\{cell(x), cdr(x) = u_1\} \vdash \mathtt{seq}(setcdr(x, u_0), [\![\{cell(x), cdr(x) = u_0\}]\!])$

Rules for Primitive Operations (c-P).

(i) $\Sigma \vdash \mathtt{seq}(\vartheta(u), [\![\Sigma]\!]) \qquad \vartheta \in \{car, cdr, atom, cell\}$

(ii) $\Sigma \vdash \mathtt{seq}(eq(u_0, u_1), [\![\Sigma]\!])$

$$\text{(iii)}\quad \frac{\Sigma \vdash \mathtt{let}\{z_0 := e_0\} \ldots \mathtt{let}\{z_n := e_n\}\mathrm{f}(z_0, \ldots, z_n, [\![\Sigma']\!], e_{n+2}, \ldots, e_m)}{\Sigma \vdash \mathrm{f}(e_0, \ldots, e_n, [\![\Sigma']\!], e_{n+2}, \ldots, e_m)}$$

for z_i fresh, $\mathrm{f} \in \mathfrak{F}_{m+1}$

Divergence rule (c-D).

$$\text{(i)}\quad \frac{\Sigma \vdash \uparrow e}{\Sigma \vdash \mathtt{seq}(e, [\![\Sigma \cup \Sigma']\!])}$$