# Program Transformations for Configuring Components

Ian A. Mason & Carolyn L. Talcott
<initials>@cs.stanford.edu

**Abstract**

In this paper we report progress in the development of methods for reasoning about the equivalence of objects with memory, and the use of these methods to describe sound operations on objects in terms of formal program transformations. We also formalize three different aspects of objects: their specification, their behavior, and their canonical representation. Formal connections among these aspects provide methods for optimization and reasoning about systems of objects. To illustrate these ideas we give a formal derivation of an optimized specialized window editor from generic specifications of its components. A new result in this paper enables one to make use of symbolic evaluation (with respect to a set of constraints) to establish the equivalence of objects. This form of evaluation is not only mechanizable, it is also generalizes the conditions under which partial evaluation usually takes place.

## 1 Overview

In [19] a general challenge for partial evaluation technology was presented and illustrated by an example using the concept of component configuration. Component description and configuration are forms of programming in the large. They provide great flexibility in code reuse and in the configuring and reconfiguring of systems. Components are related to modules in Ada, to classes in traditional object oriented languages, and to modules for traditional linkers. Component descriptions can express traditional concept such as signatures, types, interfaces, modules, and implementations, but are more general than any of these traditional notions. The flexible nature of components allows intermediate optimization stages and their derivations to be easily saved; thus increasing the opportunity for reuse of both the components and the methods used for their development.

The example presented in [19] involved developing a simple two window editor from abstract descriptions of generic components such as cursors, screens, character displays, and windows. The derivation process illustrated a variety of operations on components. Objects with free parameters were created (partial application). Messages were sent to these objects in order to attain a given internal state. Information about the context of use was specified and some simplifications based on that information were made. Objects were opened up to expose the representation of internal state, and simplifications in the representation of the combined internal state were made. Finally the free parameters were abstracted to produce the desired specialized editor constructor.

Components resulting from simple configuration have the correct behavior, but need optimization. The complexity and variety of component configurations suggests that to realize the full potential for code reuse, interactive computer aided configuration and optimization tools are needed. This paper presents a mathematical foundation for building these tools. Symbolic evaluation with respect to a set of constraints is central to our approach. It can be thought of as a form of partial evaluation, with the constraints generalizing the usual dichotomy of known–unknown.

The paper is organized as follows. §2 describes our syntactic and semantic framework. In §3 we describe various forms of program equivalence and methods for establishing them. §4 introduces the formal notions of a specification, its corresponding behavior, and the object so specified. In §5 we give several examples of these notions and their use. §6 is devoted to the formal derivation of the specialized window editor. Concluding remarks are the subject of §7.

## 2 The Framework.

Formally our language is an extension of the call-by-value lambda calculus obtained by adding primitive operations that create, access, and modify memory

cells (together with a collection of basic constants and operations on these basic constants). Our language can be thought of as 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 (*cons*, *car*, *setcar*, *eq*, etc.) but no assignment to bound variables. The reason for this choice is that it simplifies the semantics and allows one to neatly separate the functional aspects from the imperative ones.

We fix a countably infinite set of variables, $\mathbb{X}$, a countable set of atoms, $\mathbb{A}$, and a family of $n$-ary operation symbols, $\mathbb{F} = \{\mathbb{F}_n \mid n \in \mathbb{N}\}$, with $\mathbb{X}$, $\mathbb{A}$, $\mathbb{F}_n$ all pairwise disjoint. We assume $\mathbb{A}$ contains two distinct elements playing the role of booleans, `t` for *true* and `Nil` for *false*. Operations are partitioned into algebraic operations and memory operations. The unary memory operations are $\{atom, cell, car, cdr\}$ and binary memory operations are $\{eq, cons, setcar, setcdr\}$. The $n$-ary algebraic operations are functions mapping $\mathbb{A}^n$ to $\mathbb{A}$. From the given sets we define expressions, value expressions, lambda abstractions, value substitutions, and contexts.

**Definition ($\mathbb{U}, \mathbb{L}, \mathbb{E}$):** The set of value expressions, $\mathbb{U}$, the set of lambda abstractions, $\mathbb{L}$, and the set of expressions, $\mathbb{E}$, are the least sets satisfying the following equations:

$$\mathbb{U} := \mathbb{X} + \mathbb{A} + \mathbb{L}$$

$$\mathbb{L} := \lambda\mathbb{X}.\mathbb{E}$$

$$\mathbb{E} := \mathbb{U} + \mathtt{if}(\mathbb{E},\mathbb{E},\mathbb{E}) + \mathtt{app}(\mathbb{E},\mathbb{E}) + \bigcup_{n\in\mathbb{N}} \mathbb{F}_n(\mathbb{E}^n)$$

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 $\mathbb{E}$. $e\{x := e'\}$ 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 $\sigma$, $\sigma_0$, ... range over value substitutions. $e^\sigma$ 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 $\varepsilon$ 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.

In order to make programs easier to read we introduce some abbreviations. Multi-ary application and abstraction is obtained by currying, and application is represented by juxtaposition rather than explicitly writing out `app`. `let` is lambda-application (tainted sometimes by ML pattern matching). $\mathtt{seq}(e_0, \ldots, e_n)$ evaluates the expressions $e_i$ in order, returning the value of the last expression. This can be represented using `let` or `if`. We also write $null(x)$ for $eq(x, \mathtt{Nil})$. `cond` is the usual Lisp conditional, (again tainted sometimes by ML pattern matching). $\langle e_0, \ldots, e_n \rangle$ abbreviates the expression constructing a list with elements described by $e_0, \ldots, e_n$. A unary cell is the analog of an ML reference. We define operations *mk*, *get*, *set* to represent the constructor, access, and update operations on unary cells.

An operational semantics based on memory structures and a purely syntactic operational semantics for $\mathbb{E}$ are given in [11, 13]. We give a very brief outline of the syntactic semantics here, as it provides a natural basis for reasoning about program equivalence. Details may be found in [11, 13].

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. Memory contexts describe memory states and are contexts, $\Gamma$, of the form:

$$\begin{array}{l}\mathtt{let}\{z_1 := cons(\mathtt{Nil},\mathtt{Nil})\}\\ \quad\ddots\\ \quad\mathtt{let}\{z_n := cons(\mathtt{Nil},\mathtt{Nil})\}\\ \qquad\mathtt{seq}(setcar(z_1,u_1^a),\\ \qquad\quad setcdr(z_1,u_1^d),\\ \qquad\quad\vdots\\ \qquad\quad setcar(z_n,u_n^a),\\ \qquad\quad setcdr(z_n,u_n^d),\\ \qquad\quad\varepsilon)\end{array}$$

where $z_i \neq z_j$ when $i \neq j$. 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 $\Gamma'$ such that $\mathrm{Dom}(\Gamma') = \mathrm{Dom}(\Gamma) \cup \{z\}$, $\Gamma'(z) = [u_a, u_d]$, and $\Gamma'(z') = \Gamma(z')$ if $z \neq z'$.

An expression is either a value expression or decomposes uniquely into a redex placed in a reduction context. Reduction contexts, $\mathbb{R}$, identify the subexpression of an expression that is to be evaluated next, we let $R$, $R'$ range over $\mathbb{R}$. Redexes describe the primitive computation steps. A primitive step is either the application of a lambda abstraction to a value (beta reduction), branching according to whether a test value is `Nil` or not, or the application of a primitive operation.

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 $\mapsto$. Two clauses in the defi-

nition of the Single-step reduction are:

(beta) $\Gamma; R[\![\mathtt{app}(\lambda x.e, u)]\!] \mapsto \Gamma; R[\![e\{x := u\}]\!]$

(delta) $\Gamma; R[\![\delta(u_1, \ldots, u_n)]\!] \mapsto \Gamma'; R[\![u']\!]$

where in (**delta**) we assume that either $\delta$ is an $n$-ary algebraic operation, $u_1, \ldots, u_n \in \mathbb{A}^n$, $\delta(u_1, \ldots, u_n) = u'$, and $\Gamma = \Gamma'$ or $\Gamma; R[\![\delta(u_1, \ldots, u_n)]\!] \rightharpoonup \Gamma'; R[\![u']\!]$ and, for example,

$$\Gamma; R[\![cons(u_0, u_1)]\!] \rightharpoonup \Gamma\{z := [u_0, u_1]\}; R[\![z]\!]$$

$$\Gamma; R[\![setcar(z, u)]\!] \rightharpoonup \Gamma\{z := [u, u_d]\}; R[\![z]\!]$$

where in the *cons* rule $z$ is fresh, and in the *setcar* rule we assume $z \in \mathrm{Dom}(\Gamma)$ and $\Gamma(z) = [u_a, u_d]$.

A value description is a memory context together with a value expression. A description is defined just if it reduces to a value description.

# 3 Notions of equivalence

Now we define two notions of program equivalence: operational equivalence and constrained 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 [17] and the lambda-c calculus [16]. The theory of operational equivalence for the language used in this paper is presented in [13].

Constrained equivalence is a relation between sets of constraints (on memory states) 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 is a stronger relation than operational equivalence and hence is often easier to establish. A version of constrained equivalence for the first-order subset of our language was studied in detail in [9]. An inference system which is complete for zero-order terms (first-order expressions not involving recursively defined functions) is given in [10, 12, 14]. Constrained equivalence restricted to the empty set of constraints implies operational equivalence and is the same as operational equivalence in the first-order case. Constrained equivalence naturally allows reasoning by cases and permits use of a variety of induction principles.

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. Operational equivalence is a congruence relation on expressions:

$$e_0 \cong e_1 \;\rightarrow\; (\forall C \in {}^{\varepsilon}\mathbb{E})(C[\![e_0]\!] \cong C[\![e_1]\!])$$

To define constrained equivalence we need to define the the relevant notions of constraints and satisfaction. A constraint is an expression of one of the following forms: $p(u_1, \ldots, u_n)$, $\neg p(u_1, \ldots, u_n)$, $car(u_0) \simeq u_1$, $cdr(u_0) \simeq u_1$, $u_0 \simeq u_1$, and $\neg(u_0 \simeq u_1)$, where $p$ is a predicate (for example *cell* and *atom* are predicates). We let $\Sigma$, $\Sigma'$, ... denote finite sets of constraints.

A pair consisting of a memory context and value substitution $\Gamma; \sigma$ satisfies an equation $e_0 \simeq e_1$, written

$$\Gamma \models (e_0 \simeq e_1)[\sigma],$$

just if both descriptions $\Gamma; e_j^{\sigma}$ are undefined, or both evaluate to the same value description modulo production of garbage. Similarly we define the notion of a memory context and value substitution satisfying a set of constraints $\Sigma$, written

$$\Gamma \models \Sigma[\sigma],$$

for details see [10, 12, 14]. Two expressions $e_0, e_1$ are equivalent under constraints $\Sigma$, written

$$\Sigma \models e_0 \simeq e_1,$$

just if $\Gamma \models \Sigma[\sigma]$ implies $\Gamma \models (e_0 \simeq e_1)[\sigma]$ for any $\Gamma; \sigma$ (subject to simple conditions on free variables). An important consequence of this definition is that unconstrained equivalence implies operational equivalence:

$$\emptyset \models e_0 \simeq e_1 \;\rightarrow\; e_0 \cong e_1.$$

In [10, 12, 14] we introduced a formal system for proving judgements of the form

$$\Sigma \vdash \Phi,$$

where $\Sigma$ is a finite set of constraints and $\Phi$ is an assertion of the form $e_0 \simeq e_1$. One important use of the formal system is the ability to symbolically evaluate expressions with respect to a set of constraints. Indeed the ability to define a computationally adequate notion of reduction relative to a set of constraints,

$$e_0 \overset{*}{\mapsto}_{\Sigma} e_1$$

was central to the completeness proof in [10, 12, 14].

To facilitate the discussion of constrained equivalence, we define the notion of an assertion $\Phi$ holding in a context $C$ as follows:

$$C[\![e_0 \simeq e_1]\!] = C[\![e_0]\!] \simeq C[\![e_1]\!]$$

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 $\Sigma$ 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 is when $\Sigma$ is empty. To overcome this difficulty we extend the system by developing a constraint propagation logic. Here one derives assertions of the form

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

The intended meaning of an assertion of the form $\Sigma_0 \vdash C[\![\Sigma_1]\!]$, is that if $\Sigma_0$ holds when evaluation of $C[\![\ ]\!]$ begins, then $\Sigma_1$ will hold at the point in the program text where the hole appears. One consequence of the semantics of constraint propagation is that the following context propagation rule is valid.

$$\text{(CP)} \quad \frac{\Sigma_0 \vdash C[\![\Sigma_1]\!] \quad \Sigma_1 \vdash \Phi}{\Sigma_0 \vdash C[\![\Phi]\!]}$$

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

The full set of rules may be found in [15]. The rule concerning lexical binding is:

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

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

The above system is central to developing a method for establishing the equivalence of functional objects with local store (henceforth called objects). Let $\rho_j = \lambda x.e_j$ for $j < 2$. If two objects are equivalent,

$$\Gamma_0[\![\rho_0]\!] \cong \Gamma_1[\![\rho_1]\!],$$

then (assuming $x \notin \mathrm{Dom}(\Gamma_i)$)

$$\Gamma_0[\![e_0]\!] \cong \Gamma_1[\![e_1]\!].$$

The question is, when is the converse true? Under what conditions can we infer *global* equivalence from *local* equivalence? Intuitively, it must be the case that access to cells of $\Gamma_j$ can not leak out in evaluation of $\Gamma_j[\![e_j]\!]$ and it must not leak in from any external (shared) memory. To capture this intuition we first give a simple syntactic criteria that guarantees this *locality* condition. We say that that $e$ is *$\bar{z}$-local,* if the following two conditions hold.

(a) elements of $\bar{z}$ occur only as the argument to a *get*, or the first argument to a *set*, and do not occur inside a $\lambda$ abstraction (other than the body of a `let`),

(b) the free variables of $e$ not among $\bar{z}$ are examined only as atoms (no *get*s or function applications).

A set of constraints $\Sigma$ with free variables $\bar{y}$ is pure if its semantics is independent of any memory context, and if $\Sigma$ holds of some sequence of values $\bar{v}$, then each element of $\bar{v}$ is either an atom or (equivalent to) a pure object (one that mentions no memory operations).

The main result and principle tool in later sections is the following.

**Theorem (abstractable):** Suppose that

$$\begin{aligned}\Gamma = &\mathtt{let}\{z_0 := mk(v_0)\}\\ &\quad\ddots\\ &\qquad\mathtt{let}\{z_{m-1} := mk(v_{m-1})\}\\ &\qquad\quad[\![\ ]\!]\end{aligned}$$

and $\Sigma$ is a set of pure constraints, with $\mathrm{FV}(\Sigma)$ disjoint from $\mathrm{FV}(e_j)$ and

1. $e_j$ are $\bar{z}$-local for $j < 2$,
2. $\Sigma \vdash \Gamma[\![\texttt{<}\bar{z}, e_0\texttt{>} \simeq \texttt{<}\bar{z}, e_1\texttt{>}]\!]$,
3. $\Sigma \vdash \Gamma[\![\mathtt{seq}(e_0, \mathtt{let}\{v_i := get(z_i)\}_{i \le m}[\![\Sigma]\!])]\!]$.

Then

$$\Sigma \vdash \Gamma[\![\lambda x.e_0]\!] \cong \Gamma[\![\lambda x.e_1]\!].$$

The inclusion of $\bar{z}$ in 2. guarantees that the $e_j$ have the same effect on local memory as well as externally supplied memory.

# 4 Specifying Objects

We specify an object by a set of local parameters, a message parameter, and a sequence of message handlers. A message handler consists of a test function, a reply function and a list of updating functions (one for each parameter). The functions take as arguments the message and current value of the local parameters. Upon receipt of a message, the first handler whose test is true is invoked. The local parameters are updated according to the update expressions and the reply is computed by the reply function. (Evaluation of the test, updating, and reply functions should have no (visible) effect.) A specification $S$ with $k$ local parameters $\bar{x}$, message parameter $m$, and $i$th message handler with test function $t_i$, reply function $r_i$, and a updating functions $u_{i,j}$ for $1 \le j \le k$ is

written in the form:

$$S = (\bar{x})(m)[t_i(\bar{x}, m) \Rightarrow r_i(\bar{x}, m),\ u_{i,1}(\bar{x}, m),\ \ldots\ u_{i,k}(\bar{x}, m)]_{1 \le i \le n}$$

In order to make specifications concise we omit idempotent updating functions and content-free replies ($r = \texttt{Nil}$). We associate to each specification $S$ two programs: the local behavior function $beh_S$, and the canonical specified object $obj_S$. The local behavior corresponding to $S$ is purely functional. It is a closure with local parameters corresponding to those of the specification. When applied to a message, the behavior function corresponding to the updated local parameters is returned along with the reply to the message.

**Definition ($beh_S$):**

$$beh_S(\bar{x})(m) \leftarrow$$
$$\texttt{cond}[\ldots$$
$$t_i(m, \bar{x}) \Rightarrow \texttt{<}beh_S(u_{i,1}(m, \bar{x}),\ \ldots\ u_{i,k}(m, \bar{x})),\ r_i(m, \bar{x})\texttt{>}$$
$$\ldots$$
$$\texttt{t} \Rightarrow \texttt{<}beh_S(\bar{x}), \texttt{nil>}]_{1 \le i \le n}$$

The object specified by $S$ has the local parameters stored in its local memory. When applied to a message, the object updates the local parameter memory and returns only the reply.

**Definition ($obj_S$):**

$$obj_S(\bar{z})(m) \leftarrow$$
$$\texttt{let}\{x_1 := get(z_1)\} \ldots \texttt{let}\{x_k := get(z_k)\}$$
$$\texttt{cond}[\ldots$$
$$t_i(m, \bar{x}) \Rightarrow \texttt{seq}(set(z_1, u_{i,1}(m, \bar{x})),\ \ldots\ set(z_k, u_{i,k}(m, \bar{x})),\ r_i(m, \bar{x}))$$
$$\ldots$$
$$\texttt{t} \Rightarrow \texttt{nil}]_{1 \le i \le n}$$

There is a protocol transforming operation $b2o$ (behavior-to-object) that maps the behavior corresponding to $S$ to the object specified by $S$. $b2o$ allocates a cell and stores the behavior function there. When applied to a message it looks up the behavior, applies it to the message, stores the new behavior, and returns the reply. (There is also an inverse operation, but that is not needed here.) Behavior functions and objects generalize the notions of reusable and onetime streams studied in [13].

**Definition ($b2o$):**

$$b2o(beh) \leftarrow b2ox(mk(beh))$$
$$b2ox(z) \leftarrow \lambda(m)\texttt{let}\{\texttt{<}beh, r\texttt{>} := get(z)(m)\}\ \texttt{seq}(set(z, beh), r)$$

The relation between objects and behaviors, corresponding to the same specification, is captured by the following theorem.

**Theorem (b2o):**

$$b2o(beh_S(\bar{x})) \simeq \texttt{let}\{z_1 := mk(x_1)\}\ \ddots\ \texttt{let}\{z_k := mk(x_k)\}\ obj_S(\bar{z})$$

Note that it is easier to compose behaviors and reason about them than it is with the corresponding objects. Using the connections established by the abstract specification and the protocol transformation one can obtain objects corresponding to the transformed behaviors. Methods developed in [13, 15] can be extended to further simplify and optimize the resulting objects. The point is that different representations are better suited for carrying out different sorts of transformations. Thus it is important to have appropriate representations at hand and to be able to move from one representation to another in a semantically sound manner.

# 5 Component Specifications

We are working in the somewhat old fashioned world of character oriented displays and simple text editors which can manage multiple windows (ala emacs). Thus a display has a screen (two dimensional character array) of some fixed dimension (width × height), and a cursor centered at some point on the screen. A window is some subregion of a host screen, and as windows may overlap a window must have a local representation of its contents (i.e. a local display). Positions $p$ on a screen are vectors of numbers $[p_x, p_y]$ corresponding to the horizontal and vertical coordinates: $Pt = [\mathbb{N}, \mathbb{N}]$. We assume that vectors are atoms, (i.e. not cells, in particular not mutable data) and make use of the usual operations on them. In particular arithmetic operations are extended coordinate-wise to position vectors. We adopt the convention that the upper left hand corner of a screen is the origin $[0, 0]$ and $[x, y]$ is the position reached by moving $y$ down and $x$ to the right. The predicate $in(p, o, d)$ asserts that the point $p$ lies in the rectangle with origin $o$ and dimension $d$.

$$in(p, o, d) \leftarrow$$

$$(o_x \leq p_x < o_x + d_x) \wedge (o_y \leq p_y < o_y + d_y)$$

## 5.1 Cursor

A cursor of dimension $d$ is a point $p$ constrained to be on a rectangular grid of dimension $m$ — the set of points $p$ such that $in(p, o, d)$ It responds to messages: `Pos` — return the coordinates of the point; and `L`, `R`, `U`, `D` — move cursor left, right, up, down (sticking at the boundaries). For convenience we define the following operations

$$mvL(p) \leftarrow \texttt{if}(p_x \leq 0, p, p - [1, 0])$$
$$mvR(p, d) \leftarrow \texttt{if}(p_x - 1 \geq d_x, p, p + [1, 0])$$

We present the three alternate descriptions of cursors as a concrete illustration of these concepts. In later examples we will only present the most appropriate form.

**Definition (Cursor specification):**

$$curS = (d, p)(m)[$$
$$m = \texttt{Pos} \Rightarrow r = p$$
$$m = \texttt{L} \Rightarrow u_p = mvL(p)$$
$$m = \texttt{R} \Rightarrow u_p = mvR(p, d)$$
$$\ldots ]$$

**Definition (Cursor behavior):**

$$curB(d, p) \leftarrow$$
$$\lambda m.\texttt{cond}[$$
$$m = \texttt{Pos} \Rightarrow \texttt{<}curB(d, p), p\texttt{>}$$
$$m = \texttt{L} \Rightarrow \texttt{<}curB(d, mvL(p)), \texttt{nil>}$$
$$m = \texttt{R} \Rightarrow \texttt{<}curB(d, mvR(p, d)), \texttt{nil>}$$
$$\ldots$$
$$\texttt{t} \Rightarrow \texttt{<}curB(d, p), \texttt{nil>} ]$$

**Definition (Canonical Cursor object):**

$$curO(z_d, z_p) \leftarrow$$
$$\lambda m.\texttt{let}\{d := get(z_d)\}\texttt{let}\{p := get(z_p)\}$$
$$\texttt{cond}[$$
$$m = \texttt{Pos} \Rightarrow \texttt{seq}(set(z_d, d), set(z_p, p), p)$$
$$m = \texttt{L} \Rightarrow \texttt{seq}(set(z_d, d), set(z_p, mvL(p)), \texttt{Nil})$$
$$m = \texttt{R} \Rightarrow \texttt{seq}(set(z_d, d), set(z_p, mvR(p, d)), \texttt{Nil})$$
$$\ldots$$
$$\texttt{t} \Rightarrow \texttt{nil} ]$$

By applying basic laws of constrained equivalence, assuming $z_p$ and $z_d$ are distinct cells, we can simplify the cursor object description to the following.

**Definition (Simplified Cursor object):**

$$curO'(d, z_p) \leftarrow \lambda m.\texttt{let}\{p := get(z_p)\}$$
$$\texttt{cond}[$$
$$m = \texttt{Pos} \Rightarrow p$$
$$m = \texttt{L} \Rightarrow \texttt{seq}(set(z_p, mvL(p)))$$
$$m = \texttt{R} \Rightarrow \texttt{seq}(set(z_p, mvR(p, d)))$$
$$\ldots$$
$$\texttt{t} \Rightarrow \texttt{nil} ]$$

**Lemma (curob):**

$$\texttt{let}\{z_d := mk(d)\}\texttt{let}\{z_p := mk(p)\}$$
$$[\![curO(z_d, z_p) \cong curO'(d, z_p)]\!]$$

**Proof (curob):** Let

$$\Sigma[d, p] = \{Pt(d), Pt(p)\}$$
$$\Gamma_{\text{loc}} = \texttt{let}\{z_d := mk(d)\}\texttt{let}\{z_p := mk(p)\}[\![\ ]\!]$$
$$C_{\text{loc}} = \Gamma_{\text{loc}}[\![\texttt{<}z_d, z_p, [\![\ ]\!]\texttt{>}]\!]$$
$$e_0 = curO(z_d, z_p)(m)$$
$$e_1 = curO'(d, z_p)(m)$$

We note that within the context $\Gamma_{\text{loc}}$ the cells $z_d$ and $z_p$ are provably distinct. We will use the (**abstractable**) theorem. First observe that the syntactic locality conditions hold. Consequently we must show the following.

(i) $\Sigma[d, p] \vdash C_{\text{loc}}[\![e_0 \simeq e_1]\!]$

(ii) $\Sigma[d, p] \vdash$
$$\Gamma_{\text{loc}}[\![\texttt{seq}(e_0,$$
$$\texttt{let}\{d' := get(z_d)\}$$
$$\texttt{let}\{p' := get(z_p)\}[\![\Sigma[d', p']]\!]]\!]$$

The proof of (**i**) uses the rules for `let` evaluation, pushing contexts across the tests of a `cond`, and removing redundant *sets*.

To prove (**ii**) it suffices to consider each branch of the `cond`. For the messages under consideration we have the following four simple subcases.

(ii.1) $\Sigma[d, p] \vdash \texttt{seq}(pos, [\![\Sigma[d, p]]\!])$

(ii.2) $\Sigma[d, p] \vdash \texttt{seq}(set(z_p, mvL(p)),$
$$\texttt{let}\{p' := get(z_p)\}[\![\Sigma[d, p']]\!])$$

(ii.3) $\Sigma[d, p] \vdash \texttt{seq}(set(z_p, mvR(p, d)),$
$$\texttt{let}\{p' := get(z_p)\}[\![\Sigma[d, p']]\!])$$

(ii.4) $\Sigma[d, p] \vdash \texttt{seq}(\texttt{nil}, [\![\Sigma[d, p]]\!])$

$\square_{\textbf{curob}}$

## 5.2 Screen

A screen $s$ of dimension $d$ is a map from the rectangular grid of dimension $d$ to the set of printable characters, *Char*. It responds to messages: $\texttt{<Get}, p\texttt{>}$ — get the character at $p$; and $\texttt{<Set}, p, char\texttt{>}$ — set the character at $p$ to be *char*. $\sqcup$ corresponds to the blank character.

**Definition (Screen specification):**

$$
\begin{aligned}
&scrS = (d, \mu)(m)[\\
&\quad m = \texttt{<Get}, p\texttt{>} \Rightarrow r = \texttt{if}(in(p, [0,0], d), \mu(p), \sqcup)\\
&\quad m = \texttt{<Set}, p, x\texttt{>} \Rightarrow u_\mu = \texttt{if}(in(p, [0,0], d),\\
&\qquad\qquad\qquad\qquad \lambda q.\texttt{if}(p = q, x, \mu(p)),\\
&\qquad\qquad\qquad\qquad \mu)\ ]
\end{aligned}
$$

where $\mu$ maps each point in $d$ to a character.

## 5.3 Display

An abstract display is built from two components: a cursor and a screen. The messages a display responds to include: `Dim` — return the dimension; `Pos` — return the cursor position; $\texttt{<Charo}, char\texttt{>}$ — write *char* at the cursor position and move the cursor right; `Get` — return the character at the cursor position; and the cursor motion commands `L`, `R`, `U`, `D`.

**Definition (Display specification):**

$$
\begin{aligned}
&disS = (d, cur, scr)(m)[\\
&\quad m = \texttt{Dim} \Rightarrow r = d\\
&\quad m = \texttt{Pos} \Rightarrow\\
&\qquad r = \texttt{let}\{\texttt{<}cur, p\texttt{>} := cur(\texttt{Pos})\}p\\
&\qquad u_{cur} = \texttt{let}\{\texttt{<}cur, p\texttt{>} := cur(\texttt{Pos})\}cur\\
&\quad m = \texttt{Get} \Rightarrow\\
&\qquad r = \texttt{let}\{\texttt{<}cur, p\texttt{>} := cur(\texttt{Pos})\}\\
&\qquad\qquad \texttt{let}\{\texttt{<}scr, x\texttt{>} := scr(\texttt{<Get}, p\texttt{>})\}\\
&\qquad\qquad\quad x\\
&\qquad u_{cur} = \texttt{let}\{\texttt{<}cur, p\texttt{>} := cur(\texttt{Pos})\}cur\\
&\qquad u_{scr} = \texttt{let}\{\texttt{<}cur, p\texttt{>} := cur(\texttt{Pos})\}\\
&\qquad\qquad \texttt{let}\{\texttt{<}scr, x\texttt{>} := scr(\texttt{<Get}, p\texttt{>})\}\\
&\qquad\qquad\quad scr\\
&\quad m = \texttt{<Charo}, char\texttt{>} \Rightarrow\\
&\qquad u_{cur} = \texttt{let}\{\texttt{<}cur, *\texttt{>} := cur(\texttt{R})\}cur\\
&\qquad u_{scr} = \texttt{let}\{\texttt{<}cur, p\texttt{>} := cur(\texttt{Pos})\}\\
&\qquad\qquad \texttt{let}\{\texttt{<}scr, *\texttt{>} := scr(\texttt{<Set}, p, char\texttt{>})\}\\
&\qquad\qquad\quad scr\\
&\quad m = \texttt{L} \Rightarrow u_{cur} = \texttt{let}\{\texttt{<}cur, *\texttt{>} := cur(\texttt{L})\}cur\\
&\\
&\ldots\ ]
\end{aligned}
$$

The local behavior of a display is given by $disB(d, cur, scr)$ and the canonical display object is $disO(z_d, z_{cur}, z_{scr})$. Now, using (**abstractable**) we replace the cell $z_d$ by its contents, and apply the protocol transformation. As a result cells containing behaviors are replaced by objects meeting the same specifications.

**Definition (Display object with objects):**

$$
\begin{aligned}
&disO'(d, co, so) \leftarrow \lambda m.\texttt{cond}[\\
&\quad m = \texttt{Dim} \Rightarrow d\\
&\quad m = \texttt{Pos} \Rightarrow co(\texttt{Pos})\\
&\quad m = \texttt{Get} \Rightarrow so(\texttt{<Get}, co(\texttt{Pos})\texttt{>})\\
&\quad m = \texttt{<Charo}, char\texttt{>} \Rightarrow\\
&\qquad \texttt{seq}(so(\texttt{<Set}, co(\texttt{Pos}), char\texttt{>}), co(\texttt{R}), \texttt{nil})\\
&\quad m = \texttt{L} \Rightarrow co(\texttt{L})\\
&\qquad \ldots\\
&\quad \texttt{t} \Rightarrow \texttt{nil}\ ]
\end{aligned}
$$

**Theorem (display.xproto):** For $d$, $p$ ranging over points and $\mu$ ranging over character maps we have

$$
\begin{aligned}
&\texttt{let}\{z_d := mk(d))\}\\
&\quad \texttt{let}\{z_{cur} := mk(curB(d, p))\}\\
&\qquad \texttt{let}\{z_{scr} := mk(scrB(d, \mu))\}\\
&\qquad\quad disO(d, z_{cur}, z_{scr})\\
&\cong \texttt{let}\{co := curO(mk(d), mk(p))\}\\
&\quad \texttt{let}\{so := scrO(mk(d), mk(\mu))\}\\
&\qquad disO'(d, co, so)
\end{aligned}
$$

**Proof (dx):** The key step is to show that

$$
\begin{aligned}
&\texttt{let}\{z_{cur} := mk(curB(d, p))\}\\
&\quad \texttt{let}\{z_{scr} := mk(scrB(d, \mu))\}\\
&\qquad disO'(d, b2ox(z_{cur}), b2ox(z_{scr}))\\
&\cong \texttt{let}\{z_d := mk(d)\}\\
&\quad \texttt{let}\{z_{cur} := mk(curB(d, p))\}\\
&\qquad \texttt{let}\{z_{scr} := mk(scrB(d, \mu))\}\\
&\qquad\quad disO(z_d, z_{cur}, z_{scr})
\end{aligned}
$$

For this we unfold $b2ox(z)$ in the body of $disO'$, simplify, and use (**abstractable**). Let

$$
\begin{aligned}
&\Sigma[d, c, s] = \{Pt(d), CurB(c), ScrB(s)\}\\
&\Gamma_{\text{loc}} = \texttt{let}\{z_d := mk(d)\}\\
&\qquad\quad \texttt{let}\{z_{cur} := mk(c)\}\\
&\qquad\qquad \texttt{let}\{z_{scr} := mk(s)\}[\![\ ]\!]\\
&C_{\text{loc}} = \Gamma_{\text{loc}}[\![\texttt{<}z_d, z_{cur}, z_{scr}, [\![\ ]\!]\texttt{>}]\!]\\
&e_0 = disO(z_d, z_{cur}, z_{scr})(m)\\
&e_1 = disO'(d, b2ox(z_{cur}), b2ox(z_{scr}))(m)
\end{aligned}
$$

It is easy to see (after unfolding the definitions) that the syntactic locality conditions hold. Thus what we must show is the following.

(i) $\Sigma[d, c, x] \vdash C_{\mathrm{loc}}[\![e_0 \simeq e_1]\!]$

(ii) $\Sigma[d, c, s] \vdash \Gamma_{\mathrm{loc}}[\![\mathtt{let}\{r := e_0\}$
$\mathtt{let}\{d' := get(z_d)\}$
$\mathtt{let}\{c' := get(z_{cur})\}$
$\mathtt{let}\{s' := get(z_{scr})\}$
$[\![\Sigma[d', c', s']]\!]]\!]$

We push the initial `let`s into the branches of the `cond`s and consider cases on the form of $m$. For (ii) we must show that the constraint $\Sigma[d, c, s]$ propagates. Using constrained equivalence we need only show propagation for $disO'$. Here we use the following properties of $CurB$ and $ScrB$

(c.i) $\{CurB(c)\} \vdash \mathtt{let}\{z := mk(c)\}$
$\mathtt{let}\{r := b2ox(z)(m)\}$
$\mathtt{let}\{c' := get(z)\}$
$[\![\{CurB(c')\}]\!]$

(c.ii) $\{CurB(c)\} \vdash \mathtt{let}\{z := mk(c)\}$
$\mathtt{let}\{r := b2ox(z)(\mathtt{Pos})\}$
$\mathtt{let}\{c' := get(z)\}$
$[\![\{c \simeq c', Pt(r)\}]\!]$

(s.i) $\{ScrB(s)\} \vdash \mathtt{let}\{z := mk(s)\}$
$\mathtt{let}\{r := b2ox(z)(m)\}$
$\mathtt{let}\{s' := get(z)\}$
$[\![\{ScrB(s')\}]\!]$

$\square_{\mathbf{dx}}$

### 5.4 Windows

A window is a display embedded in another (host) display. It has an origin, and a local display to maintain an image of its abstract display. In addition to the normal display commands, a window also accepts the command to display itself. Since the host display may be shared by other windows, a window must be passed the current host behavior each time it is sent a message, and it must return the resulting host behavior in addition to its reply. To simplify the specifications we introduce the following auxiliary functions. $Hcharo(h, char, o, d)$ sends the character, $char$, to the host, $h$, and insures that the cursor remains in the window. For example, if the right window edge is not at the display edge and the cursor is at the right window edge, then the display cursor must be moved left after printing the character. Similarly for $Hcurl$. $Mapdis(h, dis, o)$ maps the display, $dis$, onto the host, $h$, screen displaced by the vector $o$. It returns the modified $h$. $setCur(h, p)$ sets the position of the $h$ cursor to be $p$, returning the modified $h$.

**Definition (Window specification):**

$wdoS = (o, dis)(h, m)[$
`Pos` $\Rightarrow r = \langle h, \mathtt{let}\{\langle dis, p\rangle := dis(\mathtt{Pos})\}p\rangle$
$u_{dis} = \mathtt{let}\{\langle dis, p\rangle := dis(\mathtt{Pos})\}dis$
`Get` $\Rightarrow r = \langle h, \mathtt{let}\{\langle dis, x\rangle := dis(\mathtt{Get})\}x\rangle$
$u_{dis} = \mathtt{let}\{\langle dis, p\rangle := dis(\mathtt{Get})\}dis$
`Display` $\Rightarrow r = \langle Mapdis(h, dis, o), \mathtt{nil}\rangle$
$\langle \mathtt{Charo}, char\rangle \Rightarrow$
$r = \mathtt{let}\{\langle dis, d\rangle := dis(\mathtt{Dim})\}$
$\langle Hcharo(h, char, o, d), \mathtt{nil}\rangle$
$u_{dis} = \mathtt{let}\{\langle dis, *\rangle := dis(\langle \mathtt{Charo}, char\rangle)\}$
$dis$
`L` $\Rightarrow r = \langle Hcurl(h, o), \mathtt{nil}\rangle$
$u_{dis} = \mathtt{let}\{\langle dis, *\rangle := dis(\langle \mathtt{Charo}, char\rangle)\}dis$
$\ldots]$

## 6 Specializing a window editor

A two window editor has as parameter a host display $h$, which may be shared by other applications. Our specification of a window editor uses a parameter $cn$ to remember the current window name and two window parameters $w_1$, $w_2$. It responds directly to the message `Toggle` — select the other window. Other messages are passed on to the current window. In light of the possible sharing of $h$, the abstract specification of a window editor assumes that each invocation consists of a host parameter and a message. We define the toggle operation on window names by $tog(cn) = \mathtt{if}(cn = 1, 2, 1)$.

**Definition (2 window editor):**

$2wedS = (cn, w_1, w_2)(h, m)[$
`Toggle` $\Rightarrow$
$\mathtt{let}\{cw := \mathtt{if}(nm = 1, w_2, w_1)\}$
$\mathtt{let}\{\langle cw, h\rangle := cw(h)(\mathtt{Display})\}$
$r = \langle h, \mathtt{nil}\rangle$
$u_{cn} = tog(cn)$
$u_{w_1} = \mathtt{if}(cn = 2, cw, w_1)$
$u_{w_2} = \mathtt{if}(cn = 1, cw, w_2)$
$m \Rightarrow \mathtt{let}\{cw := \mathtt{if}(cn = 1, w_1, w_2)\}$
$\mathtt{let}\{\langle cw, h, r\rangle := cw(h, m)\}$
$r = \langle h, r\rangle$
$u_{cn} = cn$
$u_{w_1} = \mathtt{if}(cn = 1, cw, w_1)$
$u_{w_2} = \mathtt{if}(cn = 2, cw, w_2)\ ]$

We now specialize by assuming the two windows do not overlap. To be concrete, we take the windows to be of dimension $[40, 24]$ and origin $[0, 0]$, $[40, 0]$, respectively, and assume the host display has dimension $[80, 24]$. We take advantage of the special situation and replace the local windows by cursors, thus eliminating display duplication, redisplaying, etc. We keep only the two cursor locations, current $cp$ and other $op$, the index of the current window $cn$ and the host $h$. The result is $2wedB'$.

**Definition (Non-overlap 2 window editor):**

$$
\begin{array}{l}
2wedB'(cn, cp, op, h) \leftarrow \lambda(m)\texttt{cond}[ \\
\quad m = \texttt{Toggle} \Rightarrow \\
\quad\quad \texttt{<}2wedB'(tog(cn), op, cp, setCur(h, op)), \texttt{nil>}) \\
\quad m = \texttt{Pos} \Rightarrow \texttt{<}2wedB'(cn, cp, op, h), \\
\quad\quad\quad\quad \texttt{if}(cn = 1, cp, cp - [40, 0])\texttt{>} \\
\quad m = \texttt{Get} \Rightarrow \texttt{let}\{\texttt{<}h, x\texttt{>} := h(\texttt{Get})\} \\
\quad\quad\quad\quad \texttt{<}2wedB'(cn, cp, op, h), x\texttt{>} \\
\quad m = \texttt{Display} \Rightarrow \texttt{<}2wedB'(cn, cp, op, h), \texttt{nil>} \\
\quad m = \texttt{<Charo}, char\texttt{>} \Rightarrow \\
\quad\quad \texttt{let}\{\texttt{<}h, *\texttt{>} := h(\texttt{<Charo}, char\texttt{>})\} \\
\quad\quad\quad \texttt{if}(cn = 1 \wedge cp_x = 39, \\
\quad\quad\quad\quad \texttt{let}\{\texttt{<}h, *\texttt{>} := h(\texttt{L})\} \\
\quad\quad\quad\quad\quad \texttt{<}2wedB'(cn, cp, op, h), \texttt{nil>} \\
\quad\quad\quad\quad \texttt{<}2wedB'(cn, cp_{charo}(cn, cp), op, h), \texttt{nil>} \\
\quad m = \texttt{L} \Rightarrow \\
\quad\quad \texttt{if}(cn = 2 \wedge cp_x = 40, \\
\quad\quad\quad \texttt{<}2wedB'(cn, cp, op, h), \texttt{nil>} \\
\quad\quad\quad \texttt{let}\{\texttt{<}h, *\texttt{>} := h(\texttt{L})\} \\
\quad\quad\quad\quad \texttt{<}2wedB'(cn, cp_l(cn, cp), op, h), \texttt{nil>} \\
\quad\quad \ldots \\
\quad \texttt{t} \Rightarrow \texttt{<}2wedB'(cn, cp, op, h), \texttt{nil>}\ ] \\
\quad \text{where} \\
cp_{charo}(cn, cp) \leftarrow \\
\quad \texttt{if}(cn = 2 \wedge cp_x = 79, cp, cp + [1, 0]) \\
cp_l(cn, cp) \leftarrow \texttt{if}(cn = 1 \wedge cp_x = 0, cp, cp + [1, 0])
\end{array}
$$

**Definition (Non-overlap hypothesis):** Let $H(h, cn, w_1, w_2, cp, op)$ be the assumption that there exist $p, p_1, p_2, \mu, \mu_1, \mu_2$ such that the conjunction of the following holds.

$$
\begin{array}{l}
in(p, [0, 0], [80, 24]) \\
in(p_1, [0, 0], [40, 24]) \\
in(p_2, [0, 0], [40, 24]) \\
in(p, [0, 0], [40, 24]) \rightarrow \mu(p) = \mu_1(p) \in Char \\
in(p, [40, 0], [80, 24]) \rightarrow \mu(p) = \mu_2(p - [40, 0]) \in Char \\
h \cong disB([80, 24], \\
\quad\quad curB([80, 24], p), \\
\quad\quad scrB([80, 24], \mu)) \\
w_1 \cong wdoB([0, 0], \\
\quad\quad disB([40, 24], \\
\quad\quad\quad curB([40, 24], p_1), \\
\quad\quad\quad scrB([40, 24], \mu_1))) \\
w_2 \cong wdoB([0, 0], \\
\quad\quad disB([40, 24], \\
\quad\quad\quad curB([40, 24], p_2), \\
\quad\quad\quad scrB([40, 24], \mu_2))) \\
cp = \texttt{if}(cn = 1, p_1, [40, 0] + p_2) \\
op = \texttt{if}(cn = 1, [40, 0] + p_2, p_1)
\end{array}
$$

Using simulation induction, we can prove the specialization is correct.

**Theorem (2wedb.opt):** Assuming $H(h, cn, w_1, w_2, cp, op)$ we have that

$$2wedB(cn, w_1, w_2, h) \cong 2wedB'(cn, cp, op, h)$$

We should also point out that many of the steps in the proof of (**2wedb.opt**) are similar to the partial evaluation techniques of Berlin [2]. In particular they make use of the fact that the shape of the data structures involved is known in advance.

To obtain a specialized 2 window editor object we first consider what specification the behavior $2wedB'$ realizes.

**Definition (Specialized 2wed specification):**

$$
\begin{array}{l}
n2wedS = (cn, cp, op, h)(m)[ \\
\quad \texttt{Toggle} \Rightarrow \\
\quad\quad u_{cn} = tog(cn) \\
\quad\quad u_{cp} = op \\
\quad\quad u_{op} = cp \\
\quad\quad u_h = \texttt{let}\{\texttt{<}h, *\texttt{>} := h(\texttt{<Setcur}, op\texttt{>})\}h) \\
\quad \texttt{Pos} \Rightarrow r = \texttt{if}(cn = 1, cp, cp - [40, 0]) \\
\quad \texttt{Get} \Rightarrow r = \texttt{let}\{\texttt{<}h, x\texttt{>} := h(\texttt{Get})\}x \\
\quad\quad\quad u_h = \texttt{let}\{\texttt{<}h, x\texttt{>} := h(\texttt{Get})\}h \\
\quad \texttt{Display} \Rightarrow \\
\quad \texttt{<Charo}, char\texttt{>} \Rightarrow \\
\quad\quad u_{cp} = \texttt{if}(cn = 1 \wedge cp_x = 39, cp, cp + [1, 0]) \\
\quad\quad u_h = \texttt{let}\{\texttt{<}h, *\texttt{>} := h(\texttt{<Charo}, char\texttt{>})\} \\
\quad\quad\quad\quad \texttt{if}(cn = 1 \wedge cp_x = 39, \\
\quad\quad\quad\quad\quad \texttt{let}\{\texttt{<}h, *\texttt{>} := h(\texttt{L})\}h, \\
\quad\quad\quad\quad\quad h) \\
\quad \texttt{L} \Rightarrow u_{cp} = \texttt{if}(cn = 2 \wedge cp_x = 40, \\
\quad\quad\quad\quad\quad cp, \\
\quad\quad\quad\quad\quad \texttt{if}(cn = 1 \wedge cp_x = 0, cp, cp + [1, 0])) \\
\quad\quad u_{op} = op \\
\quad\quad u_h = \texttt{let}\{\texttt{<}h, *\texttt{>} := \texttt{if}(cn = 2 \wedge cp_x = 40, \\
\quad\quad\quad\quad\quad\quad\quad\quad h(\texttt{L}), \\
\quad\quad\quad\quad\quad\quad\quad\quad \texttt{<}h, \texttt{nil>})\}h \\
\quad \ldots
\end{array}
$$

We call the corresponding object $2wedO'$. Now we apply the protocol transformation as in the display case to replace a host behavior by a host object. The final object construction is given below.

**Definition (Final 2 window editor object):**

$$
\begin{array}{l}
2wedO(z_{cn}, z_{cp}, z_{op}, h) \leftarrow \\
\quad \lambda(m)\texttt{let}\{cn := z_{cn}\}\texttt{let}\{cp := z_{cp}\}\texttt{let}\{op := z_{op}\} \\
\quad\quad \texttt{cond}[ \\
\quad\quad\quad m = \texttt{Toggle} \Rightarrow \\
\quad\quad\quad\quad \texttt{seq}(h(\texttt{<Setcur}, op\texttt{>}), \\
\quad\quad\quad\quad\quad set(z_{cn}, tog(cn)), \\
\quad\quad\quad\quad\quad set(z_{cp}, op), \\
\quad\quad\quad\quad\quad set(z_{op}\, cp))) \\
\quad\quad\quad m = \texttt{Pos} \Rightarrow \texttt{if}(cn = 1, cp, cp - [40, 0]) \\
\quad\quad\quad m = \texttt{Get} \Rightarrow h(\texttt{Get}) \\
\quad\quad\quad m = \texttt{Display} \Rightarrow \texttt{nil} \\
\quad\quad\quad m = \texttt{<Charo}, char\texttt{>} \Rightarrow \\
\quad\quad\quad\quad \texttt{seq}(h(\texttt{<Charo}, char\texttt{>}), \\
\quad\quad\quad\quad\quad \texttt{if}(cn = 1 \wedge cp_x = 39, \\
\quad\quad\quad\quad\quad\quad h(\texttt{L}), \\
\quad\quad\quad\quad\quad\quad set(z_{cp}, cp + [1, 0]))) \\
\quad\quad\quad m = \texttt{L} \Rightarrow \\
\quad\quad\quad\quad \texttt{if}(cn = 2 \wedge cp_x = 40, \\
\quad\quad\quad\quad\quad \texttt{nil} \\
\quad\quad\quad\quad\quad \texttt{seq}(h(\texttt{L}), \\
\quad\quad\quad\quad\quad\quad \texttt{if}(cn = 1 \wedge cp_x = 0, \\
\quad\quad\quad\quad\quad\quad\quad \texttt{nil}, \\
\quad\quad\quad\quad\quad\quad\quad \texttt{seq}(set(z_{cp}, cp + [1, 0])), \texttt{nil}))) \\
\quad\quad\quad \ldots \\
\quad\quad\quad \texttt{t} \Rightarrow \texttt{nil}\ ]
\end{array}
$$

**Lemma (2wedo.xproto):**

$$
\begin{array}{l}
\texttt{let}\{z_{cn} := mk(cn)\} \\
\quad \texttt{let}\{z_{cp} := mk(cp)\} \\
\quad\quad \texttt{let}\{z_{op} := mk(op)\} \\
\quad\quad\quad \texttt{let}\{z_h := mk(h)\} \\
\quad\quad\quad\quad 2wedO'(z_{cn}, z_{cp}, z_{op}, z_h) \\
\cong \texttt{let}\{z_{cn} := mk(cn)\} \\
\quad\quad \texttt{let}\{z_{cp} := mk(cp)\} \\
\quad\quad\quad \texttt{let}\{z_{op} := mk(op)\} \\
\quad\quad\quad\quad 2wedO(z_{cn}, z_{cp}, z_{op}, b2o(h))
\end{array}
$$

To summarize, we have transformed a 2 window editor object description obtained by gluing together component behaviors to an equivalent and much more efficient object by exploiting the information available in the resulting context.

**Theorem (2 window object):** Assuming

$$H(h, cn, w_1, w_2, cp, op)$$

$$
\begin{array}{l}
b2o(2wedB(cn, w_1, w_2, h)) \\
\quad \cong \\
\texttt{let}\{z_{cn} := mk(cn)\} \\
\quad \texttt{let}\{z_{cp} := mk(cp)\} \\
\quad\quad \texttt{let}\{z_{op} := mk(op)\} \\
\quad\quad\quad 2wedO(z_{cn}, z_{cp}, z_{op}, b2o(h))
\end{array}
$$

# 7 Concluding remarks

In Mason and Talcott [13, 15] we developed techniques for establishing the equivalence of functional programs with imperative features. In this paper we have extended these techniques and applied them to a non-trivial example in component based programming.

Components, or objects, are self-contained entities with local state. The local state of an object can only be changed by action of that object in response to a message. In our framework objects are represented as functions (closures) with mutable data bound to local variables. The techniques for reasoning about objects include: rules for establishing equivalence under a set of constraints; symbolic evaluation with respect to a set of constraints; propagation of constraints into program contexts; the method of simulation induction, used to establish the equivalence of objects.

The key new result presented in this paper is the (**abstractable**) theorem. This result enables one to make use of symbolic evaluation to establish the equivalence of objects. In the current state of development the framework treats only sequential computation. However, the techniques such as simulation induction and constraint propagation, have been designed with the goal in mind of treating objects which exist in and communicate with other objects in an open distributed system.

The example presented in this paper demonstrates the use of our techniques in the derivation of a specialized two window editor object from the high-level specifications of its generic components. Three alternate methods of specification and their formal interconnections were presented. This redundancy provides a wide range of representation of program transformations and allows flexibility to choose the most suitable representation for each kind of transformation. For example, the removal of duplicate storage and redundant updating in the two window editor was carried out in a purely functional framework. The point is that different representations are better

suited for carrying out different sorts of transformations, and one needs to have appropriate representations at hand and be able to move from one representation to another in a semantically sound manner.

The work, presented here, provides a basis for extending current methods of partial evaluation to the richer world of objects and effects. In particular it illustrates:

1. the use of symbolic evaluation, with respect to a set of constraints, to simplify programs with effects;
2. the use of constraints as a richer language for expressing partial information; and
3. the use of the abstractable theorem as a method for object specialization.

Symbolic evaluation is an important aspect of partial evaluation, and methods for symbolic evaluation in the presence of effects are necessary in order to extend partial evaluation techniques to such languages. Symbolic evaluation of an expression with respect to a set of constraints, as formalized by our inference system [14] is not only mechanizable, it is also generalizes the conditions under which partial evaluation usually takes place (cf. [3, 5, 6]). In this sense it is related the notion of generalized partial computation proposed by Futamura and Nogi [4]. Constraints generalize the usual known–unknown dichotomy of partial evaluation and can be implemented by symbolic values as in [20].

While we have by no means fully met the challenge presented in [19], we have laid the groundwork for application of partial evaluation to the problem of component manipulation and configuration. In the process we have unearthed a number of new challenges. In particular, it is clear that methods of static analysis, effect analysis and abstract interpretation (cf. [1, 3, 7, 8, 18]) can be used to great benefit in establishing conditions for the applicability of many transformation rules. We plan to investigate these matters and hope others will find the application of these methods to languages that combine functional and imperative features a stimulating challenge.

**Acknowledgements**

This research was partially supported by DARPA contract N00039-84-C-0211 and by NSF grants CCR-8718605 and CCR-8917606.

# References

[1] S. Abramsky and C. Hankin. *Abstract Interpretation of Applicative Languages.* Michael Horwood, London, 1987.

[2] Andrew A. Berlin. Partial evaluation applied to numerical computation. In *ACM Conference on Lisp and Functional Programming*, pages 139–150, 1990.

[3] A. Bondorf. Automatic autoprojection of higher order recursive equations. In *ESOP'90*, 1990.

[4] Y. Futamura and K. Nogi. Generalized partial computation. In D. Bjorner, A. P. Erschov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation.* North–Holland, 1988.

[5] N. D. Jones, C. Gomard, A. Bondorf, O. Danvy, and T. Mogensen. A self-applicable partial evaluator for the lambda-calculus. In *IEEE Computer Society 1990 International Conference on Computer Languages*, 1989.

[6] N. D. Jones, P. Sestoft, and H. Sø ndergaard. Mix: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2, 1989.

[7] J. M. Lucassen. *Types and Effects, towards the integration of functional and imperative programming.* PhD thesis, MIT, 1987. Also available as LCS TR-408.

[8] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Conference record of the 16th annual ACM symposium on principles of programming languages*, pages 47–57, 1988.

[9] I. A. Mason. *The Semantics of Destructive Lisp.* PhD thesis, Stanford University, 1986.

[10] 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.

[11] 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.

[12] I. A. Mason and C. L. Talcott. A sound and complete axiomatization of operational equivalence between programs with memory. Technical Report STAN-CS-89-1250, Department of Computer Science, Stanford University, 1989.

[13] I. A. Mason and C. L. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, to appear, 199?

[14] I. A. Mason and C. L. Talcott. Inferring the equivalence of (first-order) functional programs that mutate data. *Theoretical Computer Science*, to appear, 199?

[15] I. A. Mason and C. L. Talcott. Program transformation via constraint propagation. 199? submitted for publication.

[16] E. Moggi. Computational lambda-calculus and monads. In *Fourth Annual Symposium on Logic in Computer Science*. IEEE, 1989.

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

[18] O. Shivers. Control flow analysis in Scheme. In *Proceedings of SIGPLAN '88 Conference on Programming Language Design and Implementation*, 1988.

[19] C. L. Talcott and R. W. Weyhrauch. Partial evaluation, higher-order abstractions, and reflection principles as system building tools. In D. Bjorner, A. P. Erschov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*. North–Holland, 1988.

[20] Daniel. Weise. Graphs as intermediate representation for partial evaluation. Technical Report CSL-TR-90-421, Stanford University Computer Systems Laboratory, 1990.