Virtual Values for Language Extension

Thomas H. Austin
University of California, Santa Cruz

Tim Disney
University of California, Santa Cruz

Cormac Flanagan
University of California, Santa Cruz

Abstract

This paper focuses on *extensibility*: the ability of a programmer to use a particular language to extend the functionality and expressiveness of that language. This paper explores how to provide a useful and expressive notion of extensibility by *virtualizing* the interface between code and data. Specifically, a *virtual value* is a special kind of value. When a strict primitive operation expects a regular value but finds a virtual value in its place, that operation invokes a *trap* on the virtual value. Each virtual value contains a collection of traps, each of which is a user-defined function that describes how that operation should behave on that kind of value, and so provides a very general form of *behavioral intercession*.

This paper formalizes the semantics of virtual values, and shows how virtual values enable the definition of a variety of language extensions, each under 50 lines of code. These extensions include additional numeric types; delayed evaluation; taint tracking; contracts; revokable membranes; dynamic information flow; and symbolic execution.

Several of these topics are areas of active research. We suggest that universal proxies may enable such research to be performed by experimenting *within* a language with extensible virtual values, rather than by having to define new programming languages and implementations.

1. Introduction

Programming language design is driven by multiple, often conflicting desiderata, such as: expressiveness, simplicity, elegance, performance, correctness, and extensibility, to name just a few. This paper focuses primarily on *extensibility*: the ability of a programmer using a particular language to extend the functionality and expressiveness of that language. Extensibility is desirable on its own merits; it also helps control language complexity by allowing many aspects of functionality to be delegated to libraries, and it enables grassroots innovation, where individual programmers can extend the language rather than being restricted to particular features chosen by the language design committee.

Our starting point for language extension is the observation that language semantics typically involve interaction between code and data, where code performs various *operations* (allocation, assignment, addition, etc) on data values. Typically, the behavior of each operation is fixed or *hard-wired* by the language semantics. Thus, if a function wants to perform addition on its argument, then it must be passed a numeric value that can be understood by the built-in addition operation. Consequently, a user-defined `complex` type will not interoperate with code that uses the built-in addition operation.

This paper explores the benefits of "virtualizing" the interface between code and data values. Computer science has a strong and successful history in virtualizing other well-defined interfaces. For example, virtualizing the interface between a processor and its memory subsystem enabled innovations such as virtual memory, distributed shared memory, and memory mapped files. Virtualizing the entire processor enables multiple independent virtual machines to run on a single hardware processor, or to be migrated between processors.

We propose to enable language extension by virtualizing the entire interface between code and data. Although virtualization is often considered esoteric, with potentially complex interactions between various meta-levels, we show that the semantics of data virtualization can be elegantly captured using traditional tools of operational semantics. Specifically, we present a language that supports *virtual values*. When a primitive operation expects a regular value but finds a virtual value in its place, that operation invokes a *trap* on the virtual value. Each virtual value is simply a collection of traps, each of which is a user-defined function that describes how that operation should behave on that virtual value. The operational semantics of this language is fairly straightforward, with additional evaluation rules for invoking appropriate traps for operations on virtual values.

Virtual values enable the programmer to define entirely new kinds of values, and to have these values interoperate in a transparent manner with existing code. As might be expected, virtual values enable a variety of interesting language extensions, including:

1. Additional numeric types, such as rationals, bignums, complex numbers, or decimal floating points[1], with traditional operator syntax.
2. Lazy or delayed evaluation, with implicit forcing when a delayed value is passed to a strict operation.
3. Taint tracking.
4. Dynamic information flow analysis, which extends taint analysis to handle implicit flows.
5. Symbolic execution, where code executes on *symbolic* input values, thus providing increased test coverage by avoiding the need to pre-commit to specific test inputs [19, 14].
6. Dynamically checked contracts [9], including contracts on functions and data structures that are enforced lazily.
7. Revocable membranes, which allow two components to interact until the membrane is *revoked*, at which point no further interaction or communication is possible [22].

This work is inspired by Miller and Van Cutsem's proposal for Javascript *Catch-All Proxies* [23, 4], which provide traps for operations on functions and objects. These object proxies virtualize the interface between code and objects (including function objects), but not between code and other data values. Our work also extends prior work on reflection [20] and behavioral intercession in SmallTalk [15], in the CLOS metaobject protocol system [18], or via mirrors [2] and mirages [25]. In general, this prior work focuses on virtualizing the interface between code and *objects*, but not between code and other data values. (Section 12 contains a more detailed comparison to related work.)

Virtual values generalizes these prior ideas to virtualize the interface between code and *all data values*. This generalization provides significant benefits. In particular, it enables a number of additional interesting applications, most notably (1)–(5) from the list above. Moreover, it is particularly helpful for mainstream languages, which typically include a large collection of non-object values. Finally, virtual values generalize these prior ideas to languages, such as Scheme, that are not object oriented.

We formalize the semantics of virtual values in context of a particular dynamically typed language; however, our ideas should be generally extensible to other languages, particularly to other dynamically-typed languages that already perform tag checks.

Validating a language design feature is always difficult. Certain quantifiable aspects of language design, such as performance, are more easily validated, but are often less important than aspects such as expressiveness, consistency, elegance, and extensibility. In this paper, we aim to validate the expressiveness and extensibility benefits of virtual values by presenting a series of progressively more interesting language extensions, including: (1) delayed evaluation; (2) contracts; (3) taint analysis; (4) complex numbers; (5) revokable membranes; (6) symbolic execution; and (7) information flow analysis. Each language extension is small, under 50 lines of code, yet fairly powerful, thus validating that virtual values offer an elegant and expressive mechanism for language extension.

These extensions are nicely composable. For example, we extend the language with dynamically-checked contracts, and can use that contract extension to document interfaces in the implementation of other extensions. Our taint extension automatically tracks taint information through all code, including through the complex numbers extension or the delayed evaluation extension. Similarly, the symbolic execution extension automatically performs symbolic analysis of complex numbers or delayed thunks.

To emphasize the modularity benefits of virtual values, we briefly consider the consequences of an alternative architecture in which these extensions are implemented as part of the language itself. This approach radically complicates the language, since each extension *cross-cuts* all of the features and evaluation rules of the language. For example, the information flow and complex number extension interact in a non-trivial fashion, since we need to track how information flows through operations on complex numbers.

In contrast, virtual values enable a clear separation of concerns between the various extension modules, and provides a more coherent and extensible architecture. Composed virtual values are an instance of the Decorator Pattern [13] applied to the interface between code and data.

Virtual values are motivated by the rich proliferation of research on various kinds of wrappers and proxies, including higher-order contracts [9, 8], language interoperation via proxies [16], and hybrid and gradual typing [28, 10], and space-efficient gradual typing [29]. Virtual values may allow some of this research to be performed simply by experimenting within a language with virtual values, rather than by designing new languages and implementations.

Contributions The main contributions of this paper are as follows:

- it virtualizes the entire interface between code and data values, thus generalizing prior methods for behavioral intercession in metaobject protocols;
- it demonstrates that behavioral intercession is not restricted to objects, or to object-oriented languages;
- it presents a formal yet accessible operational semantics for virtual values;
- and it demonstrates the extensibility benefits of virtual values by implementing seven substantial language extensions: (1) delayed evaluation; (2) contracts; (3) taint analysis; (4) complex numbers; (5) revokable membranes; (6) symbolic execution; and (7) information flow.

[1] Decimal floating point numbers (IEEE 754-2008) avoids the unintuitive rounding errors of binary floating point. Our work is partly motivated by discussions within the ECMA TC39 Javascript standardization committee regarding the desire for a decimal floating point library that could support convenient operator syntax.

Figure 1: $\lambda_{\textbf{proxy}}$ **Syntax**

$e ::=$	**Expressions**
x	variable
c	constants
$\lambda x.\, e$	abstraction
$e\ e$	application
$\texttt{if}\ e\ e\ e$	conditional
$\mathit{uop}\ e$	unary operators
$e\ \mathit{bop}\ e$	binary operators
$\{\,\overline{e : e}\,\}$	record creation
$e[e]$	record lookup
$e[e] := e$	record update
$\texttt{proxy}\ e$	proxy creation
$\texttt{isProxy}\ e$	proxy predicate

$c \quad ::= n \mid s \mid \texttt{false} \mid \texttt{true} \mid \texttt{unit}$ **Constants**

$\mathit{uop} ::= - \mid\ ! \mid \texttt{isNum} \mid \texttt{isBool}$ **Unary operators**

$\quad \texttt{isFunction} \mid \texttt{isRecord} \mid \texttt{tostring} \mid \ldots$

$\mathit{bop} ::= + \mid = \mid\ != \mid \ldots$ **Binary operators**

Syntactic Sugar

$$\begin{array}{rcl}
e.\texttt{x} & \overset{\text{def}}{=} & e[\texttt{"x"}] \\
e.\texttt{x} := e' & \overset{\text{def}}{=} & e[\texttt{"x"}] := e' \\
\texttt{x} : e & \overset{\text{def}}{=} & \texttt{"x"} : e \\
\texttt{let}\ x = e_1;\ e_2 & \overset{\text{def}}{=} & (\lambda x.\, e_2)\ e_1 \\
e_1;\ e_2 & \overset{\text{def}}{=} & (\lambda x.\, e_2)\ e_1 \qquad x \notin FV(e_2) \\
\texttt{letrec}\ x = e_1;\ e_2 & \overset{\text{def}}{=} & \texttt{let}\ y = \{\};\ y.x := \theta e_1;\ \theta e_2 \\
 & & \text{where } \theta = [x := y.x] \\
e_1 \mid\mid e_2 & \overset{\text{def}}{=} & \texttt{let}\ x = e_1;\ \texttt{if}\ x\ x\ e_2 \\
e_1\ \&\&\ e_2 & \overset{\text{def}}{=} & \texttt{let}\ x = e_1;\ \texttt{if}\ x\ e_2\ x \\
\lambda.\, e & \overset{\text{def}}{=} & \lambda d.\, e \qquad d \notin FV(e) \\
f() & \overset{\text{def}}{=} & f\ \texttt{unit} \\
\texttt{assert}\ e & \overset{\text{def}}{=} & \texttt{if}\ e\ \texttt{unit}\ (\texttt{unit}\ \texttt{unit})
\end{array}$$

$$\begin{array}{l}
\overline{\texttt{private}\ x = e};\ \overline{y = e'} \\
\qquad\qquad \overset{\text{def}}{=} \\
\quad \texttt{let}\ p = \{\};\ \texttt{let}\ q = \{\};\ \overline{p.x := \theta e};\ \overline{q.y := \theta e'};\ q \\
\qquad\qquad\qquad \text{where } \theta = [\overline{x := p.x}, \overline{y := q.y}]
\end{array}$$

2. A Language With Virtual Values

We formalize the semantics of virtual values in the context of an idealized language that extends the dynamically typed λ-calculus with virtual values, as well as with mutable, extensible records (as in Javascript). For brevity, we use *proxy* as a synonym for virtual values, and so refer to the language as λ_{proxy}.

2.1 Syntax

The syntax of λ_{proxy} is summarized in figure 1. In addition to the usual abstractions ($\lambda x.\, e$), applications ($e\ e$), and variables (x) of the λ-calculus, the language also has constants (c), conditionals ($\texttt{if}\ e\ e\ e$), and unary and binary operators ($\mathit{uop}\ e$ and $e\ \mathit{bop}\ e$, respectively). Constants include numbers (n) and strings (s), as well as `unit` and boolean constants.

A *record* is mutable finite map from values to values. The language includes constructs to create ($\{\,\overline{e : e}\,\}$), lookup ($e[e]$), and update ($e[e] := e$) this map. The domain of a record is often strings, and so following Javascript we include syntactic sugar to facilitate this common case, whereby $e.\texttt{x}$ abbreviates $e[\texttt{"x"}]$, etc. A record access returns `false` by default (similar to `undefined` in Javascript) if an accessed field is not defined in a record.

A proxy value p is created by the expression $\texttt{proxy}\ e$, where e should evaluate to a *handler record* that defines a collection of *trap functions* with the following informal meanings:

`call`	::	`argument` → `result`
`getr`	::	`index` → `contents`
`geti`	::	`record` → `contents`
`setr`	::	`index` → `newcontents` → `Unit`
`seti`	::	`record` → `newcontents` → `Unit`
`unary`	::	*uop* → `result`
`left`	::	*bop* → `rightarg` → `result`
`right`	::	*bop* → `leftarg` → `result`
`test`	::	`Unit` → `Any`

The `call` trap defines how the proxy p should behave when it is used as a function and applied to a particular argument, as in ($p\ \mathit{arg}$). The `getr` and `setr` traps define the proxy's behavior when used as a record, as in $p[w]$ and $p[w] := v$, respectively. The `geti` and `seti` traps are called when the proxy p is used as a record index, as in $a[p]$ and $a[p] := v$.

The `unary` trap is invoked when a unary operator is applied to the proxy (e.g., $!p$). The specific unary operator is passed as a string argument (e.g., `"!"`), which facilitates handling all unary operations in a consistent and compact manner. (Strings play the role of enum types.)

For binary operators, the proxy could occur on the left or the right side of the operator, and each case invokes a corresponding trap (`left` or `right`), with the binary operator string and the other operand being passed as arguments. If both operands are proxies we give precedence to the left argument, and so the `right` trap is invoked only when the left operand is not a proxy. Finally, if a proxy is used in a conditional test, then the proxy's `test` trap is invoked, which should return a value to be used in that test.

Figure 1 includes the usual abbreviations for `let` and `letrec`, for the short-circuiting operators || and &&, and for defining and invoking thunks. A failing `assert` is modeled by getting stuck. To facilitate defining each language extension, we introduce a lightweight syntax for modules

$$\overline{\texttt{private}\ x = e};\ \overline{y = e'}$$

with private variables $\overline{x}$, public variables $\overline{y}$, and where all definitions can be mutually recursive. In the desugared form of this construct, the records p and q hold the private and public bindings respectively, and only the public bindings in

Figure 2: $\lambda_{\textbf{proxy}}$ **Semantics**

Runtime Syntax:

$$\begin{array}{rcll}
r & ::= & c \mid a \mid \lambda x.\, e & \text{Raw values} \\
v, w & ::= & r \mid \texttt{proxy}\ a & \text{Values} \\
e & ::= & \ldots \mid a & \text{Expressions with addresses} \\
H & ::= & \mathit{Address} \to_p (\mathit{Value} \to_p \mathit{Value}) & \text{Heaps} \\
E & ::= & \bullet\, e \mid v\, \bullet \mid \texttt{if}\ \bullet\ e\ e \mid \mathit{uop}\ \bullet \mid \bullet\ \mathit{bop}\ e \mid v\ \mathit{bop}\ \bullet \mid \texttt{proxy}\ \bullet \mid \texttt{isProxy}\ \bullet \mid & \text{Evaluation context frames} \\
 & & \bullet[e] \mid v[\bullet] \mid \bullet[e] := e \mid v[\bullet] := e \mid v[w] := \bullet \mid \{\, \overline{v : v}, \bullet : e, \overline{e : e}\, \} \mid \{\, \overline{v : v}, v : \bullet, \overline{e : e}\, \} &
\end{array}$$

Evaluation Rules:

$$\begin{array}{rcllr}
H, (\lambda x.\, e)\ v & \to & H, e[x := v] & & [\text{CALL}] \\
H, \{\, \overline{s : v}\, \} & \to & H[a := \{\, \overline{s : v}\, \}], a & a \notin \mathit{dom}(H) & [\text{ALLOC}] \\
H, a[w] & \to & H, v & w \in \mathit{dom}(H(a)), v = H(a)(w) & [\text{GET}] \\
H, a[w] & \to & H, \texttt{false} & w \notin \mathit{dom}(H(a)) & [\text{GETFALSE}] \\
H, a[w] := v & \to & H', v & H' = H[a := H(a)[w := v]] & [\text{SET}] \\
H, \mathit{uop}\ r & \to & H, \delta(\mathit{uop}, r) & & [\text{UNARYOP}] \\
H, r_1\ \mathit{bop}\ r_2 & \to & H, \delta(\mathit{bop}, r_1, r_2) & & [\text{BINARYOP}] \\
H, \texttt{if}\ r\ e_1\ e_2 & \to & H, e_1 & r \neq \texttt{false} & [\text{IFTRUE}] \\
H, \texttt{if}\ \texttt{false}\ e_1\ e_2 & \to & H, e_2 & & [\text{IFFALSE}] \\
H, \texttt{isProxy}\ (\texttt{proxy}\ a) & \to & H, \texttt{true} & & [\text{ISPROXY}] \\
H, \texttt{isProxy}\ r & \to & H, \texttt{false} & & [\text{ISNOTPROXY}] \\
 & & & & \\
H, (\texttt{proxy}\ a)\ v & \to & H, a.\texttt{call}\ v & & [\text{CALLPROXY}] \\
H, (\texttt{proxy}\ a)[w] & \to & H, a.\texttt{getr}\ w & & [\text{GETRPROXY}] \\
H, r[\texttt{proxy}\ a] & \to & H, a.\texttt{geti}\ r & & [\text{GETIPROXY}] \\
H, (\texttt{proxy}\ a)[w] := v & \to & H, (a.\texttt{setr}\ w\ v);\ v & & [\text{SETRPROXY}] \\
H, r[\texttt{proxy}\ a] := v & \to & H, (a.\texttt{seti}\ r\ v);\ v & & [\text{SETIPROXY}] \\
H, \mathit{uop}\ (\texttt{proxy}\ a) & \to & H, a.\texttt{unary}\ \texttt{"}\mathit{uop}\texttt{"} & & [\text{UNARYPROXY}] \\
H, (\texttt{proxy}\ a)\ \mathit{bop}\ v & \to & H, a.\texttt{left}\ \texttt{"}\mathit{bop}\texttt{"}\ v & & [\text{LEFTPROXY}] \\
H, r\ \mathit{bop}\ (\texttt{proxy}\ a) & \to & H, a.\texttt{right}\ \texttt{"}\mathit{bop}\texttt{"}\ r & & [\text{RIGHTPROXY}] \\
H, \texttt{if}\ (\texttt{proxy}\ a)\ e_1\ e_2 & \to & H, \texttt{if}\ (a.\texttt{test}())\ e_1\ e_2 & & [\text{TESTPROXY}] \\
 & & & & \\
H, E[e] & \to & H', E[e'] & \text{if } H, e \to H', e' & [\text{CONTEXT}]
\end{array}$$

q are exposed to the rest of the program. The substitution θ replaces references to the module-defined variables $\overline{x}$ and $\overline{y}$ with accesses to corresponding fields of p and q respectively.

2.2 Formal Semantics

Figure 2 formalizes the informal semantics outlined above. A heap H is a finite map from addresses (a) to records. A *raw value* r is a value that is not a proxy. An evaluation state H, e contains a heap and the expression being evaluated.

The rules for the evaluation relation $H, e \to H', e'$ define how to evaluate the various constructs in the language. The first collection of evaluation rules are mostly straightforward. The conditional test considers any raw value other than false as being true. As usual, the partial function δ defines the semantics of unary and binary operators (*uop* and *bop*, respectively) on raw values. For example,

$$\delta(\texttt{"="}, v_1, v_2) \overset{\text{def}}{=} \begin{cases} \texttt{true} & \text{if } v_1 \text{ and } v_2 \text{ are identical} \\ \texttt{false} & \text{otherwise} \end{cases}$$

The second collection of rules defines how traps are invoked for proxy values. For example, according to the [CALLPROXY] rule, in a function application ($f v$), if the function f is actually a proxy ($\texttt{proxy}\ a$), then the trap $a.\texttt{call}$ (or equivalently, $a[\texttt{"call"}]$) is invoked on the argument v. Similarly, on a record access $v[w]$ where $v = (\texttt{proxy}\ a)$, the trap $a.\texttt{getr}$ is applied to the record index w, via the [GETRPROXY] rule. Updating a field of a proxy invokes its `setr` trap, and assignments always return the assigned value. Using a proxy as a record index invokes its `geti` and `seti` traps via the rules [GETIPROXY] and [SETIPROXY].

For a unary operation on a proxy, the `unary` trap is invoked, with the specific unary operator being passed as a string argument. For a binary operation, the semantics first attempts to dispatch to the left proxy argument, if that is a proxy, by calling its `left` trap via the rule [LEFTPROXY]. If the left argument is a raw value but the right argument is a proxy, then that proxy's `right` trap is invoked, passing the binary operation string and the left (raw) argument.

2.3 Virtual Values and Security

Proxies allow the implementation of additional kinds of values, and so they increase the possible observable behaviors of values. For example, in the presence of proxies, `x * x` can return a negative number (e.g., when `x` is complex). Moreover, `(a.x = a.x)` could evaluate to `false`, both because `a` is a proxy whose `get` trap returns different values, or because `a.x` is a proxy that defines unusual behavior for its "=" operation, which may not be an equivalence relation.

A larger space of value behaviors does make it harder to write defensive or security-critical code. In particular, security checks that are correct under the assumption that strings are immutable may fail when passed a proxy representing mutable strings.

There is therefore some tension on how to limit the possible observable behaviors of proxies. A value consumer might want strict limits on the behavior of values (including proxy values), while a proxy creator might want maximum flexibility to introduce novel kinds of proxy behaviors. Consequently, an important design choice is what restrictions should be placed on proxy behaviors. For example, Javascript proxies cannot override the identity operator, which therefore remains an equivalence operation [4].

In λ_{proxy}, we intentionally placed no restrictions. In particular, proxies can blur the distinction between the different kinds of values (functions, constants, and records). For example, a proxy could behave both like a function and like a record, depending on how it is used. As an alternative design, proxies could be partitioned into function proxies and record proxies, which might be required to implement different kinds of traps. The special form `isProxy` allows value consumers to defend against unwanted proxy behaviors.

2.4 Virtual Values and Performance

In dynamically-typed languages, the implementation of each primitive operation typically needs to perform a *tag check* that identifies the dynamic type of each argument value. For example, Figure 3 contains a code snippet from the SpiderMonkey Javascript interpreter for performing unary minus. This code contains a fast path for handling integer values, a second fast path for doubles, and then a slow path for handling Javascript's various implicit conversions, error handling, etc. We expect that the slow path would be an ideal place for incorporating proxy handling, without introducing any additional overhead on the common fast paths. In particular, Andreas Gal has demonstrated that Javascript proxies introduce minimal overheads [4, table 2]; we hope similar results could be achieved for our more general value proxies.

A proxy typically needs a handler record with nine traps, each of which likely needs to close over the underlying value. More efficient representations are possible. For example, "`proxy` a v" could represent a proxy for the value v, where the handler a is common to many proxies, and each trap is passed the specific underlying value v each time it is invoked. This alternative representation would reduce the space required for each proxy from tens of words to perhaps three words: a header word plus slots for a and v.

Figure 3: Tag Checks in SpiderMonkey's Unary Minus

```
if (JSVAL_IS_INT( rval )&&(i=JSVAL_TO_INT(rval ))!=0) {
    // FAST PATH
    i = −i; regs . sp[−1] = INT_TO_JSVAL(i );
} else if (JSVAL_IS_DOUBLE(rval)) {
    // SECOND FAST PATH
    ···
} else {
    // SLOW PATH
    ··· // Implicit conversions
    // IDEAL SPOT FOR HANDLING PROXIES
    ··· // Error handling
}
```

In short, it appears that proxies can be implemented fairly efficiently, particular in a dynamically typed language. Further exploration of the performance implications of proxies remains for future work.

2.5 Design Principles for Reflective APIs

Bracha and Ungar's propose three design principles for reflective APIs [2], namely encapsulation, stratification, and ontological correspondence. Proxies satisfy the *principle of encapsulation*, since the proxy API is high-level and abstract, and does not expose unnecessary details regarding a particular underlying implementation of the language. Proxies also satisfy the *principle of stratification*, since there is a clear distinction between base level values (both raw values and proxies), and meta-level values, such as the handler for a proxy value. In particular, there is no way for a user of a proxy value to access the underlying handler. Evaluating `(proxy a)["unary"]` does not return the `unary` trap function of the handler `a`; instead it invokes `a`'s `get` trap on the argument `"unary"`. Finally, proxies satisfy the *principle of ontological correspondence*, since each trap handler corresponds directly to a particular operation being performed by code on a (virtual) data value.

3. Identity Proxy

To illustrate the expressiveness and extensibility benefits of universal proxies, we present a series of progressively more interesting language extensions. Each extension is only a small fragment of code, typically 10-50 lines, yet they add significant expressive power to the language.

In each language extension, we often omit punctuation such as commas or semicolons, and use indentation to clarify nesting structure, as in Haskell. For brevity, we mostly ignore error handling, and so some traps simply get stuck if their proxy is used inappropriately. For documentation pur-

Figure 4: Identity Proxy

```
identityProxy :: Any → Proxy = λx. proxy {
    call : λy.   x y
    getr : λn.   x[n]
    geti : λr.   r[x]
    setr : λn,y. x[n] := y
    seti : λr,y. r[x] := y
    unary: λo.   unaryOps[o] x
    left : λo,r. binaryOps[o] x r
    right: λo,l. binaryOps[o] l x
    test : λ.    x
}
unaryOps :: UnaryOp ⇒ Any → Any = {
    "-"    : λx. −x
    "!"    : λx. !x // negation
    isBool : λx. isBool x
    // etc for all unary ops
}
binaryOps :: BinaryOp ⇒ Any → Any → Any = {
    "+" : λx,y. x+y
    "=" : λx,y. x=y
    // etc for all binary ops
}
```

poses, each definition includes a *contract*, whose semantics we formalize (via proxies, of course) in section 5 below.

As a starting point for our series of language extensions, Figure 4 sketches a simple proxy that has no effect on program evaluation. In particular, (identityProxy x) returns a proxy in which each trap handler simply performs the appropriate operation on the underlying argument x. For unary operations, the unary trap dispatches to an auxiliary record unaryOps, which maps each unary operator string to a function that performs the corresponding operation. The left and right traps similarly dispatch to the binaryOps lookup table.

In order for identityProxy to be entirely transparent, special care needs to be taken to hide the difference between a proxy and its underlying value. In particular, identityProxy overrides the equality operation, and so "a"=(identityProxy "a") evaluates to true. Similarly, {"a":3}[identityProxy "a"] evaluates to 3 via the geti trap.

The identityProxy is *apparently* circular, since it defines each unary operation in terms of that operation itself. To illustrate how this circularity bottoms out, consider:

$$-\,(\texttt{identityProxy}\ (\texttt{identityProxy}\ 4))$$

This expression creates a proxy p_1, in which x is bound to a second proxy p_2, in which x is in turn bound to the integer 4. The "−" operator therefore the trap p_1.unary("−"), which calls unaryOps["−"](p_2), which calls a second trap p_2.unary("−"), which in turn calls unaryOps["−"](4), which finally returns -4. Thus, any apparent circularity bottoms out at the end of the proxy chain, allowing proxies to be chained together in a composable manner.

Figure 5: Lazy Evaluation Proxy

```
delay :: Thunk → Proxy = λf.
    letrec z = (λ.let r=f(); z := λ.r; r)
    proxy {
      call : λy.   z() y
      getr : λn.   z()[n]
      geti : λr.   r[z()]
      setr : λn,y. z()[n] := y
      seti : λr,y. r[z()] := y
      unary: λo.   unaryOps[o] z()
      left : λo,r. binaryOps[o] z() r
      right: λo,l. binaryOps[o] l z()
      test : λ.    z()
    }
```

4. Lazy Evaluation Proxy

The identity proxy serves to illustrate the semantics of proxies, but does not yet provide useful functionality. We next consider a more useful proxy that extends the identity proxy to provides lazy or delayed evaluation, as shown in Figure 5. The function delay takes as an argument a thunk f (a function with no arguments), and returns a proxy that behaves like the result of f, except that that result is computed lazily, when some strict operation invokes a trap on that proxy.

The function delay creates a mutable variable[2] z containing a thunk that, when called, computes f() and stores the resulting value, wrapped in a thunk, back into z. Thus, z() returns the result of f while avoiding repeated computation. Each trap then calls z() to access the result of f. In this manner, the resulting proxy causes delayed values to be implicitly forced when needed; no explicit force operations are required in the source program.

5. Contracts

A *contract* [9] is a function that mediates between two software components: the function's argument and the context that observes the function's result. As long as these two components interact appropriately, the contract behaves like the identity function;[3] if either component engages in inappropriate interaction (for example, passing a string argument when an integer is expected), the intermediating contract detects the error and halts execution.

Figure 6 shows how to implement contracts using proxies, and provides four contract constructors. By convention, we use capitalized identifiers to denote contracts, and use the subscript c to denote contract constructors.

A flat contract has the form (Flat$_c$ pred). When applied to an argument x, this contract requires that x satisfy the predicate pred. A function contract (Function$_c$ Domain Range) requires that its argument should be a func-

[2] According to the desugaring of Figure 1, letrec-bound variables are actually record fields and so are mutable.

[3] Contracts do have a small observable effect of breaking identity, since a function and its wrapped version are not considered equal.

Figure 6: Contracts

```
// Four contract constructors
Flat_c = λpred. λx. assert (pred x); x

Function_c =
  λDomain,Range.
    λx. assert (isFunction x)
        proxy {
            call: λy.  Range (x (Domain y))
            ··· // as in identityProxy
        }

Record_c =
  λcontracts.
    λx. assert (isRecord x)
        proxy {
            getr: λn.   contracts[n] (x[n])
            setr: λn,y. x[n] := (contracts[n] y)
            ··· // as in identityProxy
        }

Map_c =
  λDomain,Range.
    λx. assert (isRecord x)
        proxy {
            getr: λn.   Range (x[Domain n])
            setr: λn,y. x[Domain n] := Range y
            ··· // as in identityProxy
        }

// some useful contracts
Bool      = Flat_c (λx. isBool x)
Num       = Flat_c (λx. isNum x)
NumOrBool = Flat_c (λx. isNum x || isBool x)
Any       = Flat_c (λx. true)
Unit      = Flat_c (λx. x = unit)
Thunk     = Unit → Any
UnaryOp   = Flat_c (λx. {"-":true, ··· }[x])
BinaryOp  = Flat_c (λx. {"+":true, ··· }[x])
Proxy     = Flat_c (λx. isProxy x)
```

Syntactic Sugar

$$\begin{array}{lcl}
\texttt{Domain} \longrightarrow \texttt{Range} & \overset{\text{def}}{=} & \texttt{Function}_c\ \texttt{Domain Range} \\
\texttt{Domain} \Rightarrow \texttt{Range} & \overset{\text{def}}{=} & \texttt{Map}_c\ \texttt{Domain Range} \\
\{\{\ \overline{s : \text{Contract}}\ \}\} & \overset{\text{def}}{=} & \texttt{Record}_c\ \{\ \overline{s : \text{Contract}}\ \} \\
\multicolumn{3}{l}{\texttt{private}\ x :: C = e;\ \overline{y :: C' = e'}} \\
 & \overset{\text{def}}{=} & \texttt{let}\ p = \{\};\ \overline{\texttt{let}\ q = \{\}}; \\
 & & p.x := \theta(C\ e);\ \overline{q.y := \theta(C'\ e')};\ q \\
 & & (\text{where } \theta = [x := p.x, \overline{y := q.y}])
\end{array}$$

Figure 7: Tainting Proxy

```
private unproxy
  :: Proxy ⇒ {{ value: Untainted }} = {};

private proxify :: Untainted → Tainted = λx.
    let p = proxy {
        call : λy.    taint(x y)
        getr : λn.    taint(x[n])
        geti : λr.    taint(r[x])
        setr : λn,y. x[n] := taint(y)
        seti : λr,y. r[x] := taint(y)
        unary: λo.    taint( unaryOps[o] x)
        left : λo,r. taint(binaryOps[o] x r)
        right: λo,l. taint(binaryOps[o] l x)
        test : λ.     x
    }
    unproxy[p] := {value:x}
    p

taint :: Any → Tainted = λx.
    if (isTainted x) x (proxify x)

isTainted :: Any → Bool =
    λx. if (unproxy[x]) true false

untaint :: Any → Untainted =
    λx. if (unproxy[x]) (unproxy[x].value) x

Tainted   = Flat_c (λx.  (isTainted x)
Untainted = Flat_c (λx. !(isTainted x))
```

tion that is applied only to values satisfying the contract `Domain` and that returns only values satisfying `Range`.

Record contracts are more interesting, both because their implementation requires proxies, and because we provide two kinds of contracts, for homogeneous and heterogeneous records. Both kinds of record contracts are enforced in a lazy manner, on each access and update of the resulting proxy. A homogeneous record contract has the form `(Map`$_c$ `Domain Range)`; a record `r` satisfies this contract if each index `n` in the domain of `r` satisfies the `Domain` contract, and the corresponding value `r[n]` satisfies `Range`. A heterogeneous record contract or *map* has the form `(Record`$_c$ `contracts)`, where `contracts` is a record mapping record indices to contracts. A record `r` satisfies this contract if for each index `n` of `r`, the value `r[n]` satisfies the contract `contracts[n]`.

We use the syntax `Domain` → `Range` and `Domain` ⇒ `Range` to abbreviate function and map contracts, respectively, and {{ $\overline{s : \text{Contract}}$ }} for heterogeneous record contracts, as shown in Figure 6. We adapt the module definition syntax from Figure 1 to support contracts on module bindings. In the remainder of this paper, we use this contract syntax and definitions to document subsequent language extensions.

6. Tainting Proxy

We next apply proxies to implement taint tracking, as shown in Figure 7. The function `proxify` takes an argument `x` and returns a proxy that behaves much like `x`, in that all traps first perform the corresponding operation on `x` (much like in the identity proxy), but then taint the result.

We need to be able to untaint values, for example, after they have been appropriately sanitized. For this purpose, we maintain an unproxy record that maps each proxy value back to the original raw value. Thus, unproxy[p] is either false, if p is not a tainting proxy, or else is a record {value:x}, if p is a tainting proxy whose underlying value is x. A value is *tainted* if it is in the domain of this map and is *untainted* otherwise. The function taint uses proxify to taint any value that is not already tainted.

Based on these definitions, tainted values now propagate through all the primitive operations of the language. For example, 4 + (taint 5) evaluates to a tainted 9, that is, a tainting proxy whose underlying raw value is 9.

The unproxy table may raise some concerns about potential memory leaks, if unproxy[p] remains live even after p has been collected. The semantic of λ_{proxy}does not specify garbage collection behavior, however. In particular, λ_{proxy}records could be implemented as *ephemeron tables* [], where the entry for unproxy[p] is collected as soon as p is garbage. Alternatively, the need for unproxy could be avoided by instead using an appropriately-protected field in each proxy p that is visible only to the module itself, and not to client code.

Several languages, such as Perl, provide tainting as a builtin feature of the language implementation, which introduces additional complexity into the compiler/interpreter and runtime data representations. Proxies allow this complexity to be isolated into a small extension module.

7. Additional Numeric Types

An often-requested feature of a programming language is the ability for the programmer to introduce additional numeric types beyond what are provided in the underlying language implementation, and to manipulate these additional types using traditional and intuitive operator syntax. As one example, Java provides Bignums, but only as a library with awkward method invocation syntax, and it does not provide rational numbers, complex numbers, or decimal floating point.

In addition, programmers often want to use operators such as "+" to manipulate other "numeric-like" types such as arrays or multidimensional arrays. Some statically typed languages, such as C++, use static operator overloading to provide this flexibility. Proxies provide similar functionality for dynamically typed scripting languages.

Figure 8 illustrates how to extend λ_{proxy} with an additional numeric type, namely complex numbers. The private variable unproxy maintains a map from each complex number proxy to a (real,imaginary) pair. The function makeComplex takes as input the two components of a complex number, and creates a proxy p that dispatches unary and binary operations appropriately. Unary operations are dispatched using the complexUnaryOps lookup table, where each entry takes real and imaginary arguments. For binary operations, the left trap first decomposes the right argument y (which should be a ordinary number or a complex number) into its real and imaginary parts, and then dispatches to the appropriate function in the complexBinOps table. Note that unproxy[y] returns false if y is not a complex number proxy, and so the short circuit operator "||" conveniently provides the desired functionality. The right trap is simpler, since its left argument is never complex. Finally, when a complex number is used as a record index, its geti and seti traps use pair as unique index.

Figure 8: Complex Proxy

```
private unproxy :: Proxy ⇒ {{ real: Num, img: Num }}
  = {}

private complexUnaryOps
   :: UnaryOp ⇒ Num → Num → Any
= {
  "-"       : λr,i. makeComplex (−r) (−i)
  tostring : λr,i. (tostring r)+"+"+(tostring i)+"i"
  ...
}

private complexBinOps
   :: BinaryOp ⇒ Num → Num → Num → Num → Any
= {
  "+" : λr1,i1,r2,i2. makeComplex (r1+r2) (i1+i2)
  "=" : λr1,i1,r2,i2. (r1=r2) && (i1=i2)
  ...
}

makeComplex :: Num → Num → Complex
= λr,i.
  let pair = {real:r, img:i}
      p = proxy {
        unary: λo.   complexUnaryOps[o] r i
        left : λo,y. let z = unproxy[y] ||
                              { real: y, img: 0 }
                     complexBinOps[o] r i z.real z.img
        right: λo,y. complexBinOps[o] y 0 r i
        geti : λr.   r[pair]
        seti : λr,y. r[pair] := y
        test : λ.    true // all Complex are non−false
      }
  unproxy[p] := pair
  p

isComplex :: Any → Bool
  = λx. if (unproxy[x]) true false

i :: Complex = makeComplex 0 1

Complex = Flat_c isComplex
```

Our example implementation exports the variable i, from which client code can conveniently construct arbitrary complex numbers, for example "1.0 + (1.0 * i)".

Note that proxies are not a "silver bullet" for compositionality. In particular, proxies use a double dispatch convention for overloading binary operators. Consequently, two independent proxy-based extensions, say Complex and Rational, may not be composable, since neither implementation knows how to add a complex and a rational number.

Generic functions, as in CLOS [18] and elsewhere, provide more flexibility but with some additional complexity.

8. Dynamic Units of Measure

Type systems have been proposed to track *units of measure*, such as meters or seconds (for example, [**?**]). Figure 9 shows how to track units dynamically via proxies. The contract `UNum` describes a number, possibly wrapped in a chain of proxies, where each proxy includes a unit of measure (a string, such as `"second"`) and an integer index. This proxy chain is kept in lexicographic ordering of units by the function `makeUNum`. Unary and binary operators on `UNum`s propagate down the proxy chain to the underlying numbers, provided the units are appropriately compatible. In particular, `"+"` requires that its arguments have identical units by calling the function `(dropUnit u i r)`, which ensures that the right argument `r` has the unit `u` with index `i`, and returns the unwrapped version of `r`. The units module then exports a single binding, `makeUnit`, which can then be used by client code to create and use their desired units of measure, as in:

```
let meter  = makeUnit "meter"
let second = makeUnit "second"
let g = 9.81 * meter / second / second
```

9. Revokable Membranes

Figure 10 describes how to implement revokable, identity-preserving membranes, which provide unavoidable transitive interposition between two software components [22]. The two components can communicate via the membrane in a transparent manner, but cannot share true references, only proxies to references. Consequently, once the membrane is revoked, no further communication is possible between the two components (unless of course there is a side channel for communication, for example via a global mutable variable).

We refer to the components on each side of the membrane as the `true` and `false` components, respectively. When an object passes from the `true` component to the `false` component, it is wrapped in a proxy. When that proxy gets passed back to the `true` component, we wish to remove that proxy wrapper in order to preserve object identity. For this purpose, we maintains two maps, `unproxy[true]` and `unproxy[false]`, where `unproxy[true]` maps from references known to the `true` component to corresponding references in the `false` component, and `unproxy[false]` is the corresponding inverse map.

The function `switch` passes a value `s` from the `src` component to the other component (`!src`). Constants are passed without being wrapped, as they cannot encode or contain an object reference. Since proxies can masquerade as constants, we also need to check that `s` is not a proxy. Note that `isProxy` is a special form and not a unary operator, and so it cannot be trapped; it always reveals the true nature of a proxy, which is critical for reasoning about the security guarantees provided by code such as membranes.

Figure 9: Dynamic Units of Measure

```
private unproxy
:: Proxy ⇒ {{ unit: String, index: Int, value: UNum }}
  = {}

private makeUNum :: String → Int → UNum → UNum
= λu,i,n.
  let p = unproxy[n];
  if (p && u = p.unit)        // avoid duplicates
    makeUNum u (i + p.index) p.value
  else if (p && u < p.unit) // keep proxies ordered
    makeUNum p.u p.index (makeUNum u i p.value)
  else // add this unit to proxy chain
    let p = proxy {
      // no call, getr, geti, setr, seti traps
      unary: λo.   unitUnaryOps[o] u i n
      left : λo,r. unitLeftOps [o] u i n r
      right: λo,l. unitRightOps[o] u i n l
      test : λ.    n // ignore units in test
    }
    unproxy[p] := { unit: u, index: i, value: n }
    p

private unitUnaryOps
:: UnaryOp ⇒ Strng → Int → UNum → Any = {
  "-"      : λu,i,n. makeUNum u i (−n)
  tostring : λu,i,n. (tostring n)+" "+u+"^"+i
  ...
}

private unitLeftOps
:: BinaryOp ⇒ String → Int → UNum → Any → Any = {
  "*": λu,i,n,r. makeUNum u i (n ∗ r)
  "/": λu,i,n,r. makeUNum u i (n / r)
  "+": λu,i,n,r. makeUNum u i (n + (dropUnit u i r))
  "-": λu,i,n,r. makeUNum u i (n − (dropUnit u i r))
  "=": λu,i,n,r. hasUnit u i r && n = (dropUnit u i r)
}

private unitRightOps // left arg never a proxy
:: BinaryOp ⇒ String → Int → UNum → Any → Any = {
  "*": λu,i,n,l. makeUNum u i    (l ∗ n)
  "/": λu,i,n,l. makeUNum u (−i) (l / n)
  "+": λu,i,n,l. assert false // unit mismatch
  "-": λu,i,n,l. assert false // unit mismatch
  "=": λu,i,n,l. false        // unit mismatch
}

private hasUnit :: String → Int → UNum → Bool
 = λu,i,n.
    let p = unproxy[n]
    p != false && u = p.unit && i = p.index

private dropUnit :: String → Int → UNum → UNum
 = λu,i,n.
    assert (hasUnit u i n)
    unproxy[n].value

makeUnit :: String → UNum = λu. makeUNum u 1 1

UNum = Flat_c (λx. unproxy[x] || isNum x)
```

Figure 10: Revokable Identity-Preserving Membranes

```
private unproxy :: Bool ⇒ Any ⇒ Any
                = { true: {}, false: {} }

private revoked :: Bool = false

private isConstant :: Any → Bool
  = λx. (isNum x) || (isBool x) || (isString x)

private switch :: Bool → Any → ConstantOrProxy
= λsrc,s.
  if (revoked) (assert false) unit
  if ( (isConstant s) && !(isProxy s) )
     s
     (unproxy[src][s] ||
      let send = switch src
      let rcv  = switch (!src)
      let p    = proxy {
             call : λy.   send (s (rcv y))
             getr : λn.   send (s [rcv n])
             geti : λr.   send ((rcv r)[s])
             setr : λn,y. s[rcv n]   := rcv y
             seti : λr,y. (rcv r)[s] := rcv y
             unary: λo.   send (unaryOps[o] s)
             left : λo,r. send (binaryOps[o] s (rcv r))
             right: λo,l. send (binaryOps[o] (rcv l) s)
             test : λ.    if (s) true false
          }
       unproxy[ src][s] := p
       unproxy[!src][p] := s
       p)

membrane :: Any → Any = switch true

revoke = λ. (revoked := true)

ConstantOrProxy = Flat_c (λx. isConstant x || isProxy x)
```

In the case where s is not a constant, if unproxy[src] already contains an entry for s, then that is returned. Otherwise, we introduce the functions send and rcv for sending and receiving values from the component src, and create a new proxy p that transitively performs the appropriate wrapping in its various traps. Finally, the two maps unproxy[true] and unproxy[false] are updated to record the relation between s and p, and then p is returned.

Note that we implement all traps, and not just the get, set, and call traps, to support situations where, for example, s might be a complex number proxy. That complex number proxy would get wrapped in an additional membrane proxy; both kinds of language extension are compatible and compositional.

10. Symbolic Execution

Traditional testing requires first precommiting to a specific (or *concrete*) test input, and then observing the behavior of the system under test (SUT) on that input. In general, it can be quite difficult to choose the appropriate test inputs to generate good branch coverage. Symbolic execution provides a method for achieving greater test coverage by exploring the behavior of the SUT on an initially undetermined *symbolic* input instead [19, 14]

Typically, symbolic execution is performed via a specialized symbolic interpreter, or by appropriately instrumenting the program via source or bytecode level rewriting. Figure 11 shows how, in a language with universal proxies, a standard execution engine (an interpreter, compiler, or JIT) can be adapted to also perform symbolic execution, simply by designing proxies that appropriately capture the behavior of symbolic input values, and which record program operations on those symbolic values.

Figure 11: Symbolic Execution

```
// Assume a symbolic library implements this interface
SymExp        =  Flat_c is_SE
is_SE         :: Any → Bool
SE_var        :: Unit → SymExp
SE_constant   :: NumOrBool → SymExp
SE_unary      :: UnaryOp  → SymExp → SymExp
SE_binary     :: BinaryOp → SymExp → SymExp → SymExp
SE_constrain  :: SymExp → Bool → Unit
SE_sat        :: SymExp → Bool

private unproxy :: Proxy ⇒ {{ symexp: SymExp }} = {}

private toSymExp :: Any → SymExp = λx.
    if unproxy[x]
       unproxy[x].symexp
       (assert (isNum x || isBool x); SE_constant x)

private toSymProxy :: SymExp → SymProxy = λse.
  let p = proxy {
    unary : λo.   toSymProxy(SE_unary o se)
    left  : λo,r. toSymProxy(SE_binary o se (toSymExp r))
    right : λo,l. toSymProxy(SE_binary o (toSymExp l) se)
    test  : λ.
      let trueOk  = SE_sat (SE_binary "!=" se false);
      let falseOk = SE_sat (SE_binary "="  se false);
      let choice = if (trueOk && falseOk)
                        heuristicallyPickBranch()
                        trueOk
      SE_constrain
         (SE_binary (if choice "!=" "=") se false);
      choice
  }
  unproxy[p] := { symexp: se }
  p

private heuristicallyPickBranch :: Unit → Bool = ···

symbolicValue :: Unit → SymProxy = λ.toSymProxy(SE_var())

SymProxy = Flat_c (λx. if (unproxy[x]) true false)
```

The first nine lines of that figure describe an interface to a library for performing symbolic reasoning about numeric and boolean variables and operations. (This library might be implemented on top of a standard SMT solver such as Simplify [6] or Z3 [5].) A *symbolic expression* (`SymExp`) is either a symbolic variable (`SE_var`), a symbolic constant (`SE_constant`), or is generated by performing unary or binary operations on symbolic expressions (`SE_unary` and `SE_binary`). Thus, a symbolic expression is essentially a tree with operators on internal nodes and symbolic variables and constants at the leafs. The symbolic library also maintains a collection of *constraints*, where the function call (`SE_constrain se`) adds an additional constraint that the symbolic expression `se` must hold. Finally, the function call (`SE_sat se`) checks if symbolic expression `se` is satisfiable in the context of the current constraints, that is, if the current constraint set plus the additional constraint `se` is satisfiable.

Using this symbolic expression library, we then generate *symbolic proxies* that can be passed to the software under test. The variable `unproxy` maps from symbolic proxies to the underlying symbolic expression. The function `toSymExp` map program values to a corresponding symbolic expression, either by looking up the `unproxy` table (for symbolic proxies) or by calling `SE_constant` (for numeric and boolean constants).

The function `toSymProxy` converts a symbolic expression `se` to a symbolic proxy `p`, where the `unary`, `left`, and `right` traps of `p` perform the appropriate operations on `se` to yield a new symbolic expression that is then recursively wrapped in a symbolic proxy. Thus, for example, if X is an `SE_var`, then the expression `(1 + (2 * toSymProxy(X)))` evaluates to a symbolic proxy containing a symbolic expression representing "`1 + (2 * X)`".

The interesting case is in the `test` trap, at which point execution can no longer be entirely symbolic, since it must commit to executing one of the two possible branches. The trap first determines the possible values for the test expression `se`. If both paths are possible, then one path is chosen via the function `heuristicallyPickBranch` in a heuristic manner (for example, to maximize branch or path coverage). Once the branch `choice` is chosen, the trap handler calls `SE_constrain` to record the new constraint on `se`, and then returns `choice` to the [TESTPROXY] rule.

The symbolic execution module then exports a single thunk, `symbolicValue`, which can be used to generate symbolic inputs for testing the target software.

For brevity, this symbolic evaluation proxy is rather simplified: it omits error checking and does not support symbolic record indices (no `geti` or `seti` traps). Nevertheless, this simple implementation illustrates the key ideas and provides useful benefits over concrete execution.

As an example, suppose we wanted to test a function `sort` that takes as input an array, and we have a function `checkSorted` for checking that the resulting array is indeed sorted. (As in Javascript, an array is represented as a record whose indices are consecutive integers.) We could test `sort` by applying it to sample inputs, as in

```
checkSorted (sort { 0:15, 1:10, 2:20 });
checkSorted (sort { 0:14, 1:11, 2:30 });
```

Unfortunately, this particular test suite is rather ill-chosen. Assuming `sort` is implemented by repeated comparisons (rather than by bucket sorting), then both test inputs will execute the same code path through `sort`. In general, it is quite difficult to manually choose sufficient test inputs to exercise all code paths.

Symbolic values allow us to avoid *pre-committing* to specific test inputs, and instead to heuristically refine the chosen symbolic inputs *on-the-fly* to execute desired code paths. In particular, we can instead test `sort` by evaluating:

```
   checkSorted (sort ({ 0: symbolicValue(),
                        1: symbolicValue(),
                        2: symbolicValue() })
```

By evaluating the above expression repeatedly and configuring `heuristicallyPickBranch` to execute a different path on each iteration, we can exhaustively test `sort` on all possible arrays of length 3. For most sort implementations, this approach terminates after just six iterations.

Symbolic execution is compatible with other proxy extensions: for example, we can generate symbolic complex numbers by evaluating:

```
symbolicValue() + (symbolicValue() * i)
```

11. Dynamic Information Flow Analysis

We next consider how to implement a dynamic information flow analysis via proxies. Whereas taint analysis only tracks *explicit* flows via assignments, information flow analysis extends taint analysis to also track *implicit* flows that communicate information via the program counter.

To facilitate tracking implicit flows, we extend the semantics of proxy. Specifically, we replace the [TESTPROXY] rule so that it passes thunks for the then and else branches to the appropriate trap (in a manner reminiscent of SmallTalk's `if:` method [15]). To avoid confusion, we refer to the new trap as `branch`, which now replaces `test`:

$$H, \texttt{if}\ (\texttt{proxy}\ a)\ e_1\ e_2 \quad\rightarrow\quad H, a.\texttt{branch}\ (\lambda.\ e_1)\ (\lambda.\ e_2)$$

Even with this semantic extension for proxies, tracking implicit flows is still rather tricky. In particular, code whose execution is conditional on a private value could assign to a public variable `x`. Unfortunately, the naive approach of upgrading `x` to private when such an assignment occurs is not sufficient, since the value of `x` also reflects private information *even when the conditional assignment is not executed.* Prior work introduced the *No-Sensitive-Upgrade* check [30, 1], which forbids assigning to a public variable from within code conditional on private information.

The *No-Sensitive-Upgrade* check may be rather restrictive. A recently proposed *split-process* alternative is to execute two copies of the program simultaneously [7], one on

Figure 12: Information Flow Proxy

```
1  private LOW = 0; private HIGH = 1; private BOTH = 2;
2  private pc   :: Int = BOTH
3
4  private unproxy
5  :: Proxy ⇒ {{ low:SingleValue, high:SingleValue }} = {}
6
7  new :: Record → Proxy = λx.
8    proxy {
9      setr : λn,y.
10       switch pc
11         case BOTH: x[n] := y
12         case LOW : x[n] := splitValue y x[n]
13         case HIGH: x[n] := splitValue x[n] y
14     ··· // as in identityProxy
15 }
16
17 splitValue :: Any → Any → MultiValue = λlo,hi.
18   let lo = (if (unproxy[lo]) (unproxy[lo].low)  lo)
19   let hi = (if (unproxy[hi]) (unproxy[hi].high) hi)
20   if (lo = hi)
21      hi
22      (let combine = λf.
23       switch pc
24         case LOW : (f lo)
25         case HIGH: (f hi)
26         case BOTH:
27           splitValue (fluid−let pc := LOW  in (f lo))
28                      (fluid−let pc := HIGH in (f hi))
29       let p = proxy {
30         call   : λy.   combine (λx. x y)
31         getr   : λn.   combine (λx. x[n])
32         geti   : λr.   combine (λx. r[x])
33         setr   : λn,y. combine (λx. x[n] := y)
34         seti   : λr,y. combine (λx. r[x] := y)
35         unary  : λo.   combine (λx. unaryOps [o] x)
36         left   : λo,r. combine (λx. binaryOps[o] x r)
37         right  : λo,l. combine (λx. binaryOps[o] l x)
38         branch: λt,e. combine (λx. if (x) t() e())
39       }
40       unproxy[p] := { low: lo, high: hi }
41       p)
42
43 MultiValue  = Flat_c (λx. if (unproxy[x]) true false)
44 SingleValue = Flat_c (λx. if (unproxy[x]) false true)
```

$$\begin{aligned}&\texttt{fluid-let}\ x := e_1\ \texttt{in}\ e_2\\ \overset{\text{def}}{=}\ &\texttt{let}\ y = x;\ x := e_1;\ \texttt{let}\ r = e_2;\ x := y;\ r\end{aligned}$$

$$\begin{aligned}&\texttt{switch}\ x\ \overline{\texttt{case}\ e : e'}\\ \overset{\text{def}}{=}\ &(\texttt{if}\ (x = e_1)\ e'_1\ (\texttt{if}\ (x = e_2)\ e'_2 \ldots \texttt{unit}))\end{aligned}$$

the real private inputs, and one on dummy public inputs. Multiple processes do introduce performance overheads and other complexities, and computation that is independent of the private data is redundantly performed by both processes.

To overcome this redundancy, we propose a *split-value semantics* that conceptually performs two evaluations, but which splits evaluations in a controlled, lazy manner: see Figure 12. Pottier and Simonet [27] used a similar semantics as a proof technique for their static information flow type system. We adapt their ideas as a dynamic analysis.

Specifically, the value of `(splitvalue` lo hi`)` contains both a public and private value. The variable `pc` $\in$ $\{$`LOW`, `HIGH`, `BOTH`$\}$ tracks whether the current computation is on low or high confidentiality data, or on both simultaneously. If `pc=LOW`, then any operation on `(splitvalue` lo hi`)` actually manipulates lo instead (line 28). If `pc=BOTH`, then any use of this split value will fork separate computations on lo and hi (lines 30–32). In more detail, `(splitvalue` lo hi`)` returns a proxy `p` with the behavior outlined above, where each trap calls `combine` passing an argument function `f` that performs the operation for that trap. The function `combine` then decides whether to invoke `f` once or twice, and which arguments (lo or hi) to pass to `f`, depending on `pc`.

Handling assignment statements is still rather subtle, since an assignment in a private context should only update the private portion of the assigned variable. To trap assignments, we require that newly-allocated records be passed to the function `new` (via the prefix "`new` $\{\ldots\}$"), and the client code should use the resulting proxy instead of the original record, to ensure that all assignments go through the proxy. (This process can be enforced by load-time rewriting.)

The `new` proxy then makes sure that each assignment then updates the appropriate component of the assigned location `x[n]`, depending on `pc`. If `pc=BOTH`, then the entire location is assigned; if `pc=LOW`, then a new split value is created, with the assigned value `y` in its low component and the previous high value of `x[n]` in its high component.

12. Related Work

As discussed in the introduction, this work is inspired by Miller and Van Cutsem's proposal for Javascript *Catch-All Proxies* [23, 4], which provide traps for operations on functions and objects. Catch-all proxies essentially provide our `get`, `set`, and `call` traps, plus additional traps for other Javascript-specific functionality. Our work extends these catch-all proxies with the `unary`, `left`, `right`, `test`, `geti` and `seti` traps, which enables interesting new applications such as symbolic execution and information flow. Our presentation re-uses Miller and Van Cutsem's terminology of *proxy*, *handler*, and *trap*.

SmallTalk [15] demonstrated the benefits of pure object oriented programming, in which all data values (including numbers) are objects, and all operations are via method calls. This pure object architecture provides a high degree of flexibility, potentially with some performance overhead. Smalltalk supports the definition of proxy objects that implement the `doesNotUnderstand:` method and that delegate to an underlying object, a technique called *intercession*. Mirages [25] provide a robust interface for intercession in AmbientTalk, a related language. The language E also supports similar proxies [24]. These languages are pure in the sense that all values are objects, and so many of the extensions that we propose could also be implemented in these languages. This paper clarifies that this degree of extensibility is not restricted to pure object languages; it can also be achieved in mainstream languages that include many non-object values, and in languages that are not object oriented.[4]

The Scheme dialect Racket [11] provides *chaperones*, which provide user-defined wrappers or proxies for procedures, structures, hash tables, vectors, and boxes, and are primarily intended to support contracts. Racket intentionally does not provide chaperones for primitive types, because of the concern that these would limit optimizations, although trace-based compilation [12] may ameliorate this concern.

CLOS provides a very flexible *metaobject protocol* [18], which provides the ability to inspect and modify the behavior of parts of the object runtime system. From a MOP perspective, universal proxies make primitives such as "+" into generic functions, where the language implementation "knows" the behavior of primitives such as addition on constants, but where primitive operations are dispatched to trap handlers for proxy objects. Unlike CLOS, virtual values do not require that the source language be class oriented.

We observe that a handler record is a metaobject representing the behavior of a proxy object. By limiting this construct to proxy objects, we sacrifice some expressiveness. Conversely, however, we gain more control over access to the handler record, since the proxy encapsulates its handler, satisfying Bracha and Ungar's principle of encapsulation [2]. In contrast, many MOP designs are more open to tampering with the object's behavior.

Aspect-oriented programming (AOP) [17] is closely related to MOP research. AOP research focuses on *cross-cutting concerns* that span multiple components of a system. As one example, aspects have been used to enforce fine-grained security policies in browsers [21]. Like AOP languages, proxies enable the developer to insert code at different *pointcuts*. These pointcuts are limited to proxy objects, which somewhat limits the power, but we feel that it may also improve the readability of the code.

In a language with a rich static type system, the "trap dispatch" operations could be resolved statically, for example via Haskell's [26] type classes. This static type based approach provides stronger correctness guarantees and improved performance over dynamically dispatched proxies, but at a cost of more conceptual complexity and some decrease in flexibility. Overall, proxies seem best suited to providing extensibility in languages whose static type systems that are less rich than Haskell, or in dynamically typed scripting languages. Also, type classes such as Haskell's `Num` class virtualize some language operations; virtual values generalize this idea to all language operations.

13. Discussion and Future Work

The language extension examples of Sections 3–10 suggest that virtual values provide an flexible language extension mechanism, and raises several topics for future work.

An important next step is getting more experience on the implementation of virtual values in a real programming language, and in the implementation of proxy extensions themselves. A larger language may require a broader proxy API than in the λ_{proxy} calculus, and we believe the λ_{proxy} calculus may facilitate the design of such larger proxy APIs.

Virtual values may benefit from different compilation techniques. For example, there is no guarantee that the result of an addition operation is a number, since it instead may be a proxy. Trace-based compilation may provide separate highly optimized code paths for the number and proxy cases.

Finally, the introduction of virtual values significantly changes the denotational semantics of the language, since values now admit additional user-defined behavior, and suggests that further study of the resulting denotational structure is required. In particular, a full abstraction result [3] for λ_{proxy} might prove particularly helpful in deciding how to restrict our proxy API for security and program verification, while still providing flexibility for language extensions.

Acknowledgements We thank David Herman, Tom Van Cutsem, and Mark Miller for valuable comments on an earlier draft of this paper.

References

[1] T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *PLAS*, 2009.

[2] G. Bracha and D. Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *OOPSLA*, pages 331–344, 2004.

[3] R. Cartwright, P.-L. Curien, and M. Felleisen. Fully abstract semantics for observably sequential languages. *Inf. Comput.*, 111(2):297–401, 1994.

[4] T. V. Cutsem and M. S. Miller. Proxies: Design principles for robust object-oriented intercession APIs. In *Dynamic Languages Symposium*, 2010.

[5] L. M. de Moura and N. Bjørner. Z3: An efficient smt solver. In *TACAS*, pages 337–340, 2008.

[6] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[7] D. Devriese and F. Piessens. Noninterference through secure multi-execution. *Security and Privacy, IEEE Symposium on*, 0:109–124, 2010.

[8] R. B. Findler and M. Blume. Contracts as pairs of projections. In *International Symposium on Functional and Logic Programming*, pages 226–241, 2006.

[9] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *Proceedings of the International Conference on Functional Programming*, pages 48–59, 2002.

[10] C. Flanagan. Hybrid type checking. In *Symposium on Principles of Programming Languages*, pages 245 – 256, 2006.

[4] From another perspective, proxies can be considered a form of object, since they carry their own behavior, even though there is no `this` binding. In this sense, a language with non-object values can be extended, by introducing proxies, into a pure object language in which all operations can be dynamically dispatched to a proxy.

[11] M. Flatt and PLT. Reference: Racket. Technical Report PLT-TR2010-1, PLT Inc., June 7, 2010. `http://racket-lang.org/tr1/`.

[12] A. Gal, B. Eich, M. Shaver, D. Anderson, B. Kaplan, G. Hoare, D. Mandelin, B. Zbarsky, J. Orendorff, M. Bebenita, M. Chang, M. Franz, E. Smith, R. Reitmaier, and M. Haghighat. Trace-based just-in-time type specialization for dynamic languages. In *Conference on Programming Language Design and Implementation*, 2009.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, Boston, MA, 1995.

[14] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Conference on Programming Language Design and Implementation*, pages 213–223, 2005.

[15] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[16] K. E. Gray, R. B. Findler, and M. Flatt. Fine-grained interoperability through mirrors and contracts. In *OOPSLA*, pages 231–245, 2005.

[17] G. Kiczales. Aspect-oriented programming. *ACM Comput. Surv.*, page 154, 1996.

[18] G. Kiczales, J. D. Rivieres, and D. G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, July 1991.

[19] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[20] P. Maes. Concepts and experiments in computational reflection. *SIGPLAN Not.*, 22(12):147–155, 1987.

[21] L. A. Meyerovich, A. P. Felt, and M. S. Miller. Object views: Fine-grained sharing in browsers. In *Proceedings of the WWW 2010, Raleigh NC, USA*, 2010.

[22] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[23] M. S. Miller and T. V. Cutsem. Catch-all proxies. `http://wiki.ecmascript.org/doku.php?id=harmony:proxies`.

[24] M. S. Miller, E. D. Tribble, and J. Shapiro. Concurrency among strangers: Programming in E as plan coordination. In *In Trustworthy Global Computing, International Symposium, TGC 2005*, pages 195–229. Springer, 2005.

[25] S. Mostinckx, T. V. Cutsem, S. Timbermont, E. G. Boix, É. Tanter, and W. D. Meuter. Mirror-based reflection in AmbientTalk. *Softw., Pract. Exper.*, 39(7):661–699, 2009.

[26] Paul Hudak and Simon Peyton-Jones and Philip Wadler (eds.). Report on the programming language Haskell: A non-strict, purely functional language version 1.2. *SIGPLAN Notices*, 27(5), 1992.

[27] F. Pottier and V. Simonet. Information flow inference for ML. *Transactions on Programming Languages and Systems*, 25(1):117–158, 2003.

[28] J. Siek and W. Taha. Gradual typing for objects. In *European Conference on Object Oriented Programming*, pages 2–27, 2007.

[29] J. G. Siek and P. Wadler. Threesomes, with and without blame. In *POPL*, pages 365–376, 2010.

[30] S. A. Zdancewic. *Programming languages for information security*. PhD thesis, Cornell University, 2002.