Virtual Values for Language Extension

Thomas H. Austin Tim Disney Cormac Flanagan
University of California, Santa Cruz

Abstract

This paper focuses on *extensibility*, the ability of a programmer using a particular language to extend the expressiveness of that language. This paper explores how to provide an interesting notion of extensibility by virtualizing the interface between code and data. A *virtual value* is a special value that supports behavioral intercession. When a primitive operation is applied to a virtual value, it invokes a *trap* on that virtual value. A virtual value contains multiple traps, each of which is a user-defined function that describes how that operation should behave on that value.

This paper formalizes the semantics of virtual values, and shows how they enable the definition of a variety of language extensions, including additional numeric types; delayed evaluation; taint tracking; contracts; revokable membranes; and units of measure. We report on our experience implementing virtual values for Javascript within an extension for the Firefox browser.

Categories and Subject Descriptors D.3.1 [*Formal Definitions and Theory*]: Semantics

General Terms Languages, Reliability, Security.

Keywords Proxies, Metaobject Protocols, Behavioral Intercession.

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 designer.

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. The behavior of each operation is typically *hardwired* by the language semantics. 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.

Computer science has a strong history of virtualizing various 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 virtual machines to run on a single hardware processor, or to migrate between processors.

This paper explores the benefits of "virtualizing" the interface between code and data values. 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 a particular operation should behave on that virtual value.

Although virtualization is often considered esoteric, with complex interactions between various meta-levels, we show that the semantics of data virtualization can be elegantly captured using the standard tools of operational semantics. This operational semantics is mostly straightforward, with additional evaluation rules for invoking traps for operations on virtual values. We formalize this semantics of virtual values in the context of a particular dynamically typed language; however, our ideas should be generally extensible to other languages.

We believe that virtual values provide a rather useful notion of language extensibility. Of course, validating a language design feature is always difficult. In this paper, we aim to validate the expressiveness and extensibility benefits of virtual values by illustrating the kinds of language extensions that they enable. These extensions include:

1. Additional numeric types, such as rationals, bignums, complex numbers, or decimal floating points[1], with traditional operator syntax.
2. Units of measure (meters, seconds, etc).
3. Lazy or delayed evaluation, with implicit forcing when a delayed value is passed to a strict operation.
4. Taint tracking.
5. Dynamically checked contracts [6], including contracts on functions and data structures that are enforced lazily.
6. Revocable membranes, which allow two components to interact until the membrane is revoked, after which further interaction is forbidden [17].

Each language extension is powerful yet small (the complete code is included in the paper), thus illustrating that virtual values offer an elegant and expressive mechanism for language extension.

These extensions are nicely composable. For example, we extend the language with contracts, and use that contract extension to document other extensions. Our taint extension automatically tracks taint information through all code, including through the complex numbers extension or the delayed evaluation extension.

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 may cross-cut the other features and evaluation rules of the language. For example, the taint tracking and complex number extension would interact in a non-trivial fashion, since we would need to track how taint information flows through operations on complex numbers.

In contrast, virtual values enable a clear separation of concerns between the various extension modules, and provide a coherent and extensible architecture. Composed virtual values are essentially an instance of the Decorator Pattern [10], which is a fairly general pattern that can be applied to any interface, but in our experience it is particularly powerful when applied to the widely-used interface between code and data.

1.1 Related Work

This work is inspired by Miller and Van Cutsem's proposal for Javascript *Catch-All Proxies* [18, 4], which provide traps for operations on functions and objects. These object proxies virtualize the interface between code and objects (including function objects). Analogous functionality has been provided in other languages, including via Racket's *chaperones* [8].

Virtual values generalizes these prior ideas to virtualize the interface between code and *all data values*, including primitive values such as integers. This generalization enables additional applications, including applications (1)–(5) from the list above, and may prove helpful for mainstream languages, which typically include a large collection of non-object values.

SmallTalk [11] demonstrated the benefits of pure object-oriented programming, in which all data values are objects, and all operations (including addition and conditional tests) are method calls. Smalltalk supports the definition of proxy objects that implement the `doesNotUnderstand:` method and that delegate to an underlying object, a technique called *behavioral intercession*. This pure object architecture provides flexibility and partially virtualizes the interface between code and data, since many operations are performed via dynamically-dispatched method calls. However, Smalltalk and related pure object languages such as AmbientTalk [20], E [19], and Python typically do not provide the full set of required traps, and in particular often omit the `geti` and `seti` traps (discussed below) that are essential for full virtualization.

This paper extends the virtualization provided by pure object languages, and moreover demonstrates that extensibility is not restricted to pure object languages: virtual values enable similar extensibility in languages that are not object oriented, or that are only partially object oriented and which include non-object values.

Language extensibility has been the target of a rich body of prior research. For example, CLOS provides a very flexible *metaobject protocol* [15], which gives the ability to inspect and modify the behavior of the object runtime system, often in a very general manner. In comparison to CLOS, virtual values provides a focused mechanism for changing the language semantics at a *per-value* granularity, which is well-suited for the kinds of language extensions that we address.

Aspect-oriented programming (AOP) [14] 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 [16]. Virtual values share similar motivations to AOP, and both enable the developer to insert code at different *point-cuts*, but using virtual values these point-cuts are chosen dynamically (based on where virtual values are used) rather than statically (as in weaving-based approaches to AOP).

In a language with a rich static type system, the "trap dispatch" operations on virtual values could be resolved statically, e.g. via Haskell's [22] type classes. This static type based approach provides stronger correctness guarantees and improved performance over virtual values, but at a

[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.

cost of more conceptual complexity and some decrease in flexibility. Overall, virtual values seem best suited to providing extensibility in languages whose static type systems are less rich than Haskell, or in dynamically typed languages. Also, whereas type classes such as Haskell's `Num` class virtualize some language operations (those that manipulate `Num` values), virtual values generalize this idea to all language operations.

Contributions: The main contributions of this paper are:

- it virtualizes the entire interface between code and data values, thus providing a general mechanism for value-specific behavioral intercession;
- it clarifies that languages that are not object oriented or only partially object oriented can still enjoy the extensibility benefits of pure object languages;
- it presents an operational semantics for virtual values;
- it illustrate the extensibility benefits of virtual values by implementing six non-trivial language extensions: (1) complex numbers; (2) units of measure; (3) delayed evaluation; (4) taint analysis; (5) contracts; and (6) revokable membranes;
- and it reports on our experience implementing this design in the Firefox browser.

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 value, 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 (`if` $e\ e\ e$), and unary and binary operators (*uop* e and e *bop* e, respectively). Constants include numbers (n) and strings (s), as well as `unit` and boolean constants.

A *record* is mutable finite map from strings to values, as in Javascript. The language includes constructs to create ($\{\,\overline{s : e}\,\}$), lookup ($e[e]$), and update ($e[e] := e$) this map. Following Javascript, we include the syntactic sugar $e.$`x` to abbreviate e[`"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 `proxy` $e_1\ e_2$. Here, e_1 should be a *secret* (discussed below) and e_2 should be a *handler record* that defines nine *trap functions* with the

Figure 1: The Language λ_{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
$\{\,\overline{s : e}\,\}$	record creation
$e[e]$	record lookup
$e[e] := e$	record update
$\texttt{proxy}\ e\ e$	proxy creation
$\texttt{isProxy}\ e$	proxy predicate
$\texttt{unProxy}\ e\ e$	proxy inspection

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

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

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

Syntactic Sugar

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

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

following informal types:

`call`	::	$\mathit{argument} \to \mathit{result}$
`getr`	::	$\mathit{index} \to \mathit{contents}$
`setr`	::	$\mathit{index} \to \mathit{newcontents} \to \mathit{Unit}$
`geti`	::	$\mathit{record} \to \mathit{contents}$
`seti`	::	$\mathit{record} \to \mathit{newcontents} \to \mathit{Unit}$
`unary`	::	$\mathit{uop} \to \mathit{result}$
`left`	::	$\mathit{bop} \to \mathit{rightarg} \to \mathit{result}$
`right`	::	$\mathit{bop} \to \mathit{leftarg} \to \mathit{result}$
`test`	::	$\mathit{Unit} \to \mathit{Any}$

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 $(p\ arg)$. 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 manner.

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.

The `isProxy` construct provides a reliable mechanism to distinguish proxy and non-proxy values, since `isProxy` is a separate construct in the language and not a unary operator whose behavior can be overridden by a proxy's `unary` trap.

Each proxy value (`proxy` $a\ h$) includes a *secret* a in addition to its handler record h. The purpose of this secret is to allow a module that allocates proxies to recognize its own proxies and to distinguish them from proxies created by other (potentially adversarial) modules. In particular, code that knows the secret a can use the `unProxy` primitive to extract the underlying handler record h from the proxy, via `unProxy` a (`proxy` $a\ h$), which evaluates to h. Conversely, code that does not know the secret a cannot gain access to the underlying handler record h.

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. (Here, overline denotes a repeated sequence.) 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 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, where each record maps strings to values. A *raw value* r is a constant, an address, or a λ-expression. A *value* v is either a raw value or else `proxy` $a\ h$, where a is the secret address and h is the handler record (or possibly a proxy that behaves like a handler record). 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, the equality operator is defined as follows, and always considers λ-expressions to be distinct (due to the difficulties of testing functional equivalence).

$$\delta(\texttt{"="}, r_1, r_2) \overset{\text{def}}{=} \begin{cases} \texttt{true} & \text{if } r_1 = r_2 \text{, neither are } \lambda\text{-exprs} \\ \texttt{false} & \text{otherwise} \end{cases}$$

The `unProxy` primitive returns the handler record h only when the secret a in the proxy matches the address a passed to `unProxy`; and returns `false` otherwise.

The third collection of rules define 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 (`proxy` $a\ h$), then the trap h.`call` (or equivalently, h[`"call"`]) is invoked on the argument v. Note that h can either be a handler record, or a proxy representing a handler record; the [CALLPROXY] rule handles both cases uniformly.

On a record access $v[w]$ where v = (`proxy` $a\ h$), the trap h.`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 [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.

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

Runtime Syntax:

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

Evaluation Rules:

$$
\begin{array}{rcllr}
H, (\lambda x.\, e)\ v & \rightarrow & H, e[x := v] & & [\textsc{Call}] \\
H, \{\, \overline{s : v}\,\} & \rightarrow & H[a := \{\, \overline{s : v}\,\}], a & a \notin \mathit{dom}(H) & [\textsc{Alloc}] \\
H, a[s] & \rightarrow & H, v & s \in \mathit{dom}(H(a)), v = H(a)(s) & [\textsc{Get}] \\
H, a[s] & \rightarrow & H, \texttt{false} & s \notin \mathit{dom}(H(a)) & [\textsc{GetFalse}] \\
H, a[s] := v & \rightarrow & H', v & H' = H[a := H(a)[s := v]] & [\textsc{Set}] \\
H, \mathit{uop}\ r & \rightarrow & H, \delta(\mathit{uop}, r) & & [\textsc{UnaryOp}] \\
H, r_1\ \mathit{bop}\ r_2 & \rightarrow & H, \delta(\mathit{bop}, r_1, r_2) & & [\textsc{BinaryOp}] \\
H, \texttt{if}\ r\ e_1\ e_2 & \rightarrow & H, e_1 & r \neq \texttt{false} & [\textsc{IfTrue}] \\
H, \texttt{if}\ \texttt{false}\ e_1\ e_2 & \rightarrow & H, e_2 & & [\textsc{IfFalse}] \\
 & & & & \\
H, \texttt{isProxy}\ (\texttt{proxy}\ a\ h) & \rightarrow & H, \texttt{true} & & [\textsc{IsProxy}] \\
H, \texttt{isProxy}\ r & \rightarrow & H, \texttt{false} & & [\textsc{NotProxy}] \\
H, \texttt{unProxy}\ a\ (\texttt{proxy}\ a\ h) & \rightarrow & H, h & & [\textsc{UnProxy}] \\
H, \texttt{unProxy}\ a\ v & \rightarrow & H, \texttt{false} & v \neq (\texttt{proxy}\ a\ h) & [\textsc{UnProxyFalse}] \\
 & & & & \\
H, (\texttt{proxy}\ a\ h)\ v & \rightarrow & H, h.\texttt{call}\ v & & [\textsc{CallProxy}] \\
H, (\texttt{proxy}\ a\ h)[w] & \rightarrow & H, h.\texttt{getr}\ w & & [\textsc{GetrProxy}] \\
H, r[\texttt{proxy}\ a\ h] & \rightarrow & H, h.\texttt{geti}\ r & & [\textsc{GetiProxy}] \\
H, (\texttt{proxy}\ a\ h)[w] := v & \rightarrow & H, (h.\texttt{setr}\ w\ v);\ v & & [\textsc{SetrProxy}] \\
H, r[\texttt{proxy}\ a\ h] := v & \rightarrow & H, (h.\texttt{seti}\ r\ v);\ v & & [\textsc{SetiProxy}] \\
H, \mathit{uop}\ (\texttt{proxy}\ a\ h) & \rightarrow & H, h.\texttt{unary}\ \texttt{"}\mathit{uop}\texttt{"} & & [\textsc{UnaryProxy}] \\
H, (\texttt{proxy}\ a\ h)\ \mathit{bop}\ v & \rightarrow & H, h.\texttt{left}\ \texttt{"}\mathit{bop}\texttt{"}\ v & & [\textsc{LeftProxy}] \\
H, r\ \mathit{bop}\ (\texttt{proxy}\ a\ h) & \rightarrow & H, h.\texttt{right}\ \texttt{"}\mathit{bop}\texttt{"}\ r & & [\textsc{RightProxy}] \\
H, \texttt{if}\ (\texttt{proxy}\ a\ h)\ e_1\ e_2 & \rightarrow & H, \texttt{if}\ (h.\texttt{test}())\ e_1\ e_2 & & [\textsc{TestProxy}] \\
 & & & & \\
H, E[e] & \rightarrow & H', E[e'] & \text{if } H, e \rightarrow H', e' & [\textsc{Context}]
\end{array}
$$

Figure 3: 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. binOps[o] x r
    right: λo,l. binOps[o] l x
    test : λ.    x
}

unaryOps :: UnaryOp ⇒ Any → Any = {
    "-"    : λx. −x
    "!"    : λx. !x // negation
    isBool : λx. isBool x
    // etc for all unary ops
}

binOps :: BinaryOp ⇒ Any → Any → Any = {
    "+" : λx,y. x+y
    "=" : λx,y. x=y
    // etc for all binary ops
}
```

3. Language Extension Examples

To illustrate the expressiveness and extensibility benefits of proxies, we use the λ_{proxy} language to implement a series of interesting language extensions. Each extension is small yet adds 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 proxies simply get stuck if used inappropriately. For documentation purposes, each definition includes a *contract*, whose semantics we formalize (via proxies) in section 3.3 below.

3.1 Identity Proxy

As a starting point for our series of language extensions, figure 3 sketches a simple proxy that has no effect on program evaluation. In particular, evaluating (`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 `binOps` lookup table. Identity proxies have no need to recognize each other, and so we pass an arbitrary fresh record address as the secret to `proxy`.

Figure 4: 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. binOps[o] z() r
        right: λo,l. binOps[o] l z()
        test : λ.    z()
    }
```

The `identityProxy` may appear to be somewhat circular, since it defines each unary operation in terms of that operation itself. To illustrate how this circularity bottoms out, consider:

```
-(identityProxy (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 above therefore invokes 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, the apparent circularity bottoms out at the end of the proxy chain, allowing proxies to compose conveniently.

In order for `identityProxy` to be transparent, we need 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, the `geti` trap ensures that

```
{"a":3}[identityProxy "a"]
```

evaluates to 3.

The appendix includes a proof that the identity proxy for a value v correctly simulates the behavior of v (provided no code in the system uses the `isProxy` or `unProxy` reflection primitives). Satisfying this property required several careful design choices in our language semantics—for example, the equality operator always considers λ-expressions to be distinct.

3.2 Lazy Evaluation Extension

We next extend the identity proxy to provide more interesting functionality, namely lazy or delayed evaluation, as shown in figure 4. The function `delay` takes as an argument a thunk `f` 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. Specifically, the

Figure 5: Contract Extension

```
// 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)
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)
```

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 and no built-in support for lazy evaluation is required in the language implementation.

[2] According to the desugaring of figure 1, the `letrec`-bound variable `z` is actually a record field and so is mutable.

3.3 Contract Extension

A *contract* [6] 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; 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 5 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 that return contracts.

- 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 function that is applied only to values satisfying the contract `Domain` and that returns only values satisfying `Range`.
- We support both homogeneous and heterogeneous record contracts. A homogeneous record contract or *map* has the form `(Map`$_c$ `Domain Range)`; a record `r` satisfies this contract if each string index `s` in the domain of `r` satisfies the `Domain` contract, and the corresponding value `r[s]` satisfies `Range`. `Domain` should be a `Flat`$_c$-generated contract that preserves identity, in order for record lookups to work correctly.
- A heterogeneous record contract 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 string index `s` of `r`, the value `r[s]` satisfies `contracts[s]`.

Both kinds of record contracts are enforced in a lazy manner, on each access and update of the resulting proxy. We use the syntax `Domain` $\rightarrow$ `Range` and `Domain` $\Rightarrow$ `Range` to abbreviate function and map contracts, respectively. We adapt the module definition syntax from figure 1 to support contracts on module bindings, and use this contract syntax to document our language extensions.

$$\begin{array}{lcl}
\texttt{Domain} \rightarrow \texttt{Range} & \overset{\text{def}}{=} & \texttt{Function}_c\ \texttt{Domain Range} \\
\texttt{Domain} \Rightarrow \texttt{Range} & \overset{\text{def}}{=} & \texttt{Map}_c\ \texttt{Domain Range}
\end{array}$$

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

Figure 6: Tainting Extension

```
private secret = {}

taint :: Any → Tainted = λx.
    if (isTainted x)
        x
    else
        proxy secret {
            value: x
            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(binOps[o] x r)
            right: λo,l. taint(binOps[o] l x)
            test : λ.    x
        }

isTainted :: Any → Bool =
    λx. if (unProxy secret x) true false

untaint :: Any → Untainted =
    λx. let h = unProxy secret x
        if (h) h.value x

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

3.4 Tainting Extension

Several languages, such as Perl, provide tainting as a built-in feature of the language implementation, which introduces additional complexity into the compiler/interpreter and run-time data representations.

Proxies allow this complexity to be isolated into a small extension module, as shown in figure 6. The function `taint` takes an argument `x` and returns a proxy that behaves much like `x`, in that all traps first perform the corresponding operation on `x` but then taint the result. Each tainting proxy is marked with a secret that is kept private to the module, so that we can reliably identify tainting proxies. A value is *tainted* if it is one of these tainting proxies and is *untainted* otherwise. To untaint values (after they have been appropriately sanitized), the handler record in the tainting proxy keeps the original value in the `value` field, so that it can be later returned by `untaint`.

Based on these definitions, tainted values now propagate through all 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.

Figure 8: Revokable Membranes

```
private secret = {}

private revoked :: Bool = false

swap :: Any → Any = λx.
  assert !revoked
  if (( isNum x || isBool x || isString x)
      && !(isProxy x))
    x
  else
    let h = unProxy secret x
    if (h)
      h.value
    else
      proxy secret {
        value: x
        call : λy.   swap (x (swap y))
        getr : λn.   swap (x[swap n])
        geti : λr.   swap ((swap r)[x])
        setr : λn,y. x[swap n]   := swap y
        seti : λr,y. (swap r)[x] := swap y
        unary: λo.   swap (unaryOps[o] x)
        left : λo,r. swap (binOps[o] x (swap r))
        right: λo,l. swap (binOps[o] (swap l) x)
        test : λ.    if (x) true false
      }

revoke = λ. (revoked := true)
```

3.5 Revokable Membranes

Figure 8 describes how to implement revokable membranes, which provide unavoidable transitive interposition between two software components [17], which we call the *dry* and *wet* components. The two components can communicate via the membrane in a transparent manner, but cannot share true references (functions or records), 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).

The function `swap` passes a value `x` from one side of the membrane to the other (from dry to wet, or vice-versa). Constants are passed without being wrapped, as they cannot contain references. Since proxies can masquerade as constants, we also need to check that `x` 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.

In the case where `x` is not a constant, we next use `unProxy` to check if it is a membrane proxy p, in which case the `value` field of the handler record contains the orig-

Figure 7: Complex Number Extension

```
private secret = {}

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.
    proxy secret {
        real : r
        img  : i
        unary: λo.   complexUnaryOps[o] r i
        left : λo,y. let h = unProxy secret y
                     if (h)
                        complexBinOps[o] r i h.real h.img
                     else
                        complexBinOps[o] r i y 0
        right: λo,y. complexBinOps[o] y 0 r i
        test : λ.    true // all Complex are non-false
    }

isComplex :: Any → Bool = λx. if (unProxy secret x) true false

i :: Complex = makeComplex 0 1

Complex = Flat_c isComplex
```

inal unwrapped value v, which is then returned. Essentially, passing v across the membrane produced proxy p, and passing p back through the membrane returns the original v.

Otherwise, if `x` is not a membrane proxy, we create a new proxy that performs the appropriate wrapping in its traps, and whose `value` field records the original value `x`. For example, if `(swap` *`dry-fn`*`)` passes a function *dry-fn* from the dry to the wet component, yielding a proxy *wet-fn*, then on any application `(`*`wet-fn wet-arg`*`)`, the `call` trap of *wet-fn* first passes *wet-arg* to the dry component (via `swap`) before feeding it as the argument into *dry-fn*, and finally passes the result of *dry-fn* back to the wet component (again, via `swap`).

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, and so both extensions compose nicely.

3.6 Additional Numeric Types

An often-requested feature of a programming language is the ability to introduce additional numeric types beyond what are provided in the language implementation, and to manipulate these additional types using traditional operator syntax. In many languages, this kind of extension is difficult. For example, Java provides `Bignums`, but only as a library with awkward method invocation syntax, and it does not provide rationals, complex numbers, or decimal floating points.

Figure 7 illustrates how to extend λ_{proxy} with an additional numeric type, namely complex numbers. The function `makeComplex` takes as input the two components of a complex number, and creates a proxy that dispatches unary and binary operations appropriately. For binary operations, the `left` trap first checks if the right argument `y` is a ordinary number or a complex number. If `y` is complex, then we pass its real and imaginary components (extracted from `y`'s handler record `h`) to the appropriate function in the

Figure 9: Dynamic Units of Measure Extension

```
private secret = {}

private makeQuantity :: String → Int → Quantity → Quantity = λu,i,n.
    let h = unProxy secret n
    if (i = 0)                       // drop zero−ary unit
         n
    else if (h && h.unit = u)        // same unit, avoid duplicates
         makeQuantity u (h.index + i) h.value
    else if (h && h.unit > u)        // keep proxies ordered
         makeQuantity h.unit h.index (makeQuantity u i h.value)
    else                             // add this unit to proxy chain
         proxy secret {
             unit : u                // record the unit, index, and underlying value in the handler
             index: i
             value: n
             // 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
         }

private unitUnaryOps :: UnaryOp ⇒ String → Int → Quantity → Any = {
    "-"      : λu,i,n. makeQuantity u i (−n)
    tostring : λu,i,n. (tostring n) + " " + u + "^" + i
    ...
}
private unitLeftOps :: BinaryOp ⇒ String → Int → Quantity → Any → Any = {
    "+": λu,i,n,r. makeQuantity u i (n + (dropUnit u i r))
    "*": λu,i,n,r. makeQuantity u i (n ∗ r)
    "/": λu,i,n,r. makeQuantity u i (n / r)
    "=": λu,i,n,r. n = (dropUnit u i r)
    ...
}
private unitRightOps :: BinaryOp ⇒ String → Int → Quantity → Any → Any = {
    // left arg never a proxy
    "+": λu,i,n,l. assert false // unit mismatch
    "*": λu,i,n,l. makeQuantity u i    (l ∗ n)
    "/": λu,i,n,l. makeQuantity u (−i) (l / n)
    "=": λu,i,n,l. false        // unit mismatch
    ...
}

private dropUnit :: String → Int → Quantity → Quantity = λu,i,n.
     let h = unProxy secret n
     assert h != false && h.unit = u && h.index = i
     h.value

makeUnit :: String → Quantity = λu. makeQuantity u 1 1
Quantity = Flat_c (λx. if (isNum x || unProxy secret x) true false)
```

complexBinOps table. Otherwise we assume that y is a real number and pass 0 as the imaginary component to the complexBinOps table function. The right trap is simpler, since its left argument is never complex.

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 protocol 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 [15] and elsewhere, provide more flexibility but with some additional complexity.

3.7 Dynamic Units Of Measure

Several type systems (see for example, [13]) have been proposed to track *units of measure*, such as meters or seconds, and to avoid the confusion of units that caused the Mars Climate Orbiter mishap. We use the term *quantity* to mean a floating point number annotated with zero or more units of measure, each of which may have an associated integer *multiplicity* or index (as in second^{-2}). Thus, an example quantity is 9.81 meters second^{-2}.

Proxies provide a convenient means to track units dynamically, as illustrated in figure 9[3]. Each quantity is represented as a chain of proxies, terminating in a floating point number. As shown in lines 13–15, each proxy contains a unit of measure, an integer index, and an underlying value (the next proxy in the chain, or a floating point number). The function makeQuantity creates these proxies, ensuring that each proxy has a non-zero index, and that the proxy chain is kept in lexicographic ordering of units with at most one proxy for each unit (i.e., no duplicates).

Unary and binary operators on a Quantity 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 binding makeUnit, which can be used by client code to create desired units of measure, as in:

```
let meter  = makeUnit "meter"
let second = makeUnit "second"
let g = 9.81 * meter / second / second
g + 1 // dynamic unit mismatch error
```

[3] For simplicity, this implementation does not support *dimensions* (such as mass), but only units of measure (such as kilograms).

Figure 10: NonProxy Extension

```
private secret = {}

swap :: Any → Any = λx.
  if (isProxy x)
      // error if not our proxy
      (unProxy secret x).value
  else if (isNum x || isBool x || isString x)
      x
  else
      proxy secret {
        value: x
        call : λy.   swap (x (swap y))
        getr : λn.   swap (x[swap n])
        geti : λr.   swap ((swap r)[x])
        setr : λn,y. x[swap n]   := swap y
        seti : λr,y. (swap r)[x] := swap y
        unary: λo.   swap (unaryOps[o] x)
        left : λo,r. swap (binOps[o] x (swap r))
        right: λo,l. swap (binOps[o] (swap l) x)
        test : λ.    if (x) true false
      }
```

4. Discussion

4.1 Virtual Values, Security and Observable Behavior

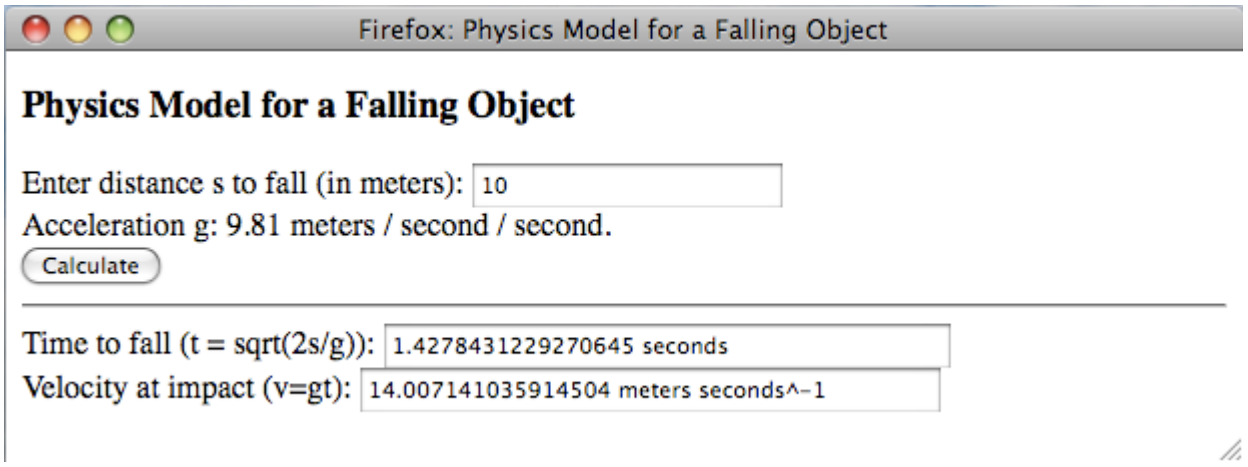
Proxies allow the implementation of additional kinds of values, and so they increase the possible observable behaviors of values. For example, in the presence of proxies, x * x can return a negative number (e.g., when x is complex). Moreover, (a.x = a.x) could evaluate to false, both because a is a proxy whose get trap returns different values, or because a.x is a proxy that defines unusual, non-reflective behavior for its "=" operation.

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 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 [4] cannot override the identity operator, which therefore remains an equivalence operation.

In λ_{proxy}, we choose to permit proxies to exhibit very general behaviors, both for simplicity and to facilitate exploration of proxy-based language extensions. To address security concerns, we provide an isProxy construct that cannot be trapped, which then allows value consumers to reliably

Figure 11: Sample web page with a Javascript evaluator running in Firefox


Firefox: Physics Model for a Falling Object

Physics Model for a Falling Object

Enter distance s to fall (in meters): 10

Acceleration g: 9.81 meters / second / second.

Time to fall (t = sqrt(2s/g)): 1.4278431229270645 seconds

Velocity at impact (v=gt): 14.007141035914504 meters seconds^-1

identify proxies and to defend against unwanted proxy behaviors.

One convenient way to protect code against unwanted proxy behavior is by using the *NonProxy* extension shown in figure 10. This extension is essentially a membrane, in that it can sit between two modules and stop values such as functions or objects from being transmitted from one module to the other, and instead sends a proxy in place of that value. The key difference is that the *NonProxy* extension forbids any attempt to send a proxy across the non-proxy membrane, and so it protects client code from values with strange, proxy-specific behavior. Thus, for example, the code fragment:

```
swap (identityProxy 4)
```

is permitted under the revokable membrane of figure 8 but not under the non-proxy membrane of figure 10.

4.2 Design Principles for Reflective APIs

Bracha and Ungar propose three design principles for reflective APIs [2], namely encapsulation, stratification, and ontological correspondence.

Proxies satisfy the *principle of encapsulation*, since the proxy API does not expose details regarding the 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 (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 h)["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.

5. Implementation: Firefox and JavaScript

In order to evaluate our approach, we extended this design for virtual values to the Javascript programming language and implemented the extended language within the Firefox browser. Our implementation leveraged the recently developed Zaphod add-on [21] for Firefox, which is based on the Narcissus [25] meta-circular Javascript interpreter. Since JavaScript is a richer language than λ_{proxy}, this extension required the introduction of the following additional traps:

- A `has` trap to determine if a proxy object has a given field.
- A `construct` trap similar to `call`, but used when the proxy is called with the `new` keyword.
- A `keys` trap to define a proxy value's behavior in a `for/in` loop.

We then implemented several of the language extension modules, including lazy evaluation, complex numbers, and units of measure. These implementations were quite straightforward and helped to validate the utility of our design. Our modified Narcissus implementation, the proxy extension modules, and the proxy test code are all available online [1]. As an illustration, figure 11 shows a web application that uses the units of measurement extension.

Performance. As a meta-circular interpreter, Narcissus does not provide a good foundation for evaluating the performance overhead of proxies. However, we believe this overhead is likely to be quite small for the common case where traps are not invoked. In a dynamically typed language, the implementation of each primitive operation typically needs to perform a *tag check* that identifies the dynamic type of each argument value. Figure 12 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

Figure 12: Tag Checks in SpiderMonkey's Unary Minus

```
if (JSVAL_IS_INT( rval )&&(i=JSVAL_TO_INT(rval ))!=0) {
    // Fast path for ints
    i = −i;
    regs . sp[−1] = INT_TO_JSVAL(i);
} else if (JSVAL_IS_DOUBLE(rval)) {
    // Second fast path for doubles
    · · ·
} else {
    · · · // Slow path for handling implicit conversions
    · · · // Ideal spot for handling proxies
    · · · // Error handling
}
```

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. Andreas Gal demonstrated that Javascript *Catch-All Proxies* introduce negligible overheads for the common case where traps are not invoked [4, table 2], and he expects that the performance overhead for virtual values would be comparably small. Of course, frequent trap invocations (*e.g.*, for complex numbers) could introduce significant overhead, and might motivate the need for additional optimization techniques. Trace-based compilation could provide highly optimized code paths that inline trap code into client code within hot loops [9].

In our current design, a proxy 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 h v" could represent a proxy for the value v, where the handler h is common to many proxies, and each trap is passed the 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 four words: a header word plus slots for a, h, and v. Overall, it appears likely that proxies can be implemented fairly efficiently, particular in dynamic languages.

6. Future Work

The language extensions presented in Section 3 provide anecdotal evidence that virtual values provide a flexible and useful language extension mechanism. In addition, our experience suggest that virtual values are fairly straightforward to incorporate into a language implementation, and that programming in the extended language remains mostly intuitive and convenient.

The introduction of virtual values does significantly change the semantics of the language, and suggests that further study of the observable equivalence relation and the denotational semantics of the language is required. In particular, a full abstraction result [3] for λ_{proxy} would clarify the space of behaviors that values (including proxy values) can exhibit, and so might be helpful in deciding how to design proxy APIs that facilitate security and program verification, while still providing flexibility to enable interesting language extensions. More research is also needed on efficient compilation and optimization techniques for virtual values.

Virtual values are motivated by the rich proliferation of research on various kinds of wrappers and proxies, including higher-order contracts [6, 5], language interoperation via proxies [12], and hybrid and gradual typing [23, 7] and space-efficient gradual typing [24]. We conjecture that virtual values may allow some of this research to be performed by experimenting within a language with virtual values, rather than by designing new languages and implementations.

Acknowledgements We thank David Herman, Tom Van Cutsem, and Mark Miller for valuable comments on an earlier draft of this paper. This work was supported by NSF grants CNS-0905650 and CCF-1116883.

References

[1] T. H. Austin. Proxy values implementation and examples. `http://slang.soe.ucsc.edu/proxy-values`, 2010.

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

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

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

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

[9] 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 *PLDI*, 2009.

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

[11] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing

Co., Inc., Boston, MA, USA, 1983.

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

[13] A. Kennedy. Relational parametricity and units of measure. In *Principles of Programming Languages*, pages 442–455, 1997.

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

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

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] Mozilla labs: Zaphod add-on for the firefox browser. `http://mozillalabs.com/zaphod`, accessed October 2010.

[22] 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.

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

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

[25] Wikipedia: Narcissus JavaScript engine. `http://en.wikipedia.org/wiki/ Narcissus_(JavaScript_engine)`, accessed October 2010.

A. Correctness of the Identity Proxy

In this appendix, we provide an illustrative proof that the identity proxy for a value v simulates the behavior of v, provided the code does not use the `isProxy` or `unProxy` primitives. In other words, the `isProxy` and `unProxy` primitives provide the only way to distinguish between a value and an identity proxy for that value. (Correctness proofs for the other language extensions are similar but somewhat more involved.)

Figure 13: Revised Identity Extesion

```
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.   { "-"    : λx. −x
                   "!"    : λx. !x // negation
                   isBool : λx. isBool x
                   // etc for all unary ops
                 }[o] x
    left : λo,r. { "+" : λx,y. x+y
                   "=" : λx,y. x=y
                   // etc for all binary ops
                 }[o] x r
    right: λo,l. { "+" : λx,y. x+y
                   "=" : λx,y. x=y
                   // etc for all binary ops
                 }[o] l x
    test : λ.    x
}
```

To allow a more direct and cleaner proof, we rewrite the identity proxy as shown in figure 13, and use

$$\texttt{HR}_v = \{\texttt{call} : \lambda y.\, v\; y, \ldots\}$$

to denote the handler record from figure 13 with x replaced by the closed value v.

Consider a value v where the heap maps an address a to the handler record $\texttt{HR}_v$, and consider a proxy value ($\texttt{proxy}\ c\ a$). Here, c is a heap address (the secret) that is irrelevant to the correctness argument. We need to show that $\texttt{proxy}\ c\ a$ simulates v in any context C, that is, that $C[\texttt{proxy}\ c\ a]$ simulates $C[v]$. For this purpose, we define a simulation relation that relates the evaluations of $C[v]$ and $C[\texttt{proxy}\ c\ a]$. The evaluation of $C[\texttt{proxy}\ c\ a]$ will execute additional proxy code that will allocate additional records in the heap. We refer to these proxy-allocated records (including $\texttt{HR}_v$) as *meta*-records, and use G to refer to the meta portion of the heap.

We first define a simulation relation $e_1 \sim_G e_2$ on expressions (with respect to a meta-heap G) as shown in figure 14. The rule [S-PROXY] states that a value v is simulated by $\texttt{proxy}\ c\ a$ provided $G(a)$ is a handler record $\texttt{HR}_{v'}$ where v is in turn simulated by v'. The remaining rules essentially provide the compatible closure of the [S-PROXY] rule, with the exception that the `isProxy` and `unProxy` primitives are not permitted (since they would distinguish between a value and its identity proxy). Note that the relation $e_1 \sim_G e_2$ is not reflexive or symmetric.

The simulation relation $H_1 \sim_G H_2$ on heaps holds if H_1 and H_2 have the same domain and for all addresses a in this

Figure 14: Expression Simulation Relation $e \sim_G e$

$$\frac{v \sim_G v' \qquad G(a) = \mathtt{HR}_{v'}}{v \sim_G \mathtt{proxy}\ c\ a} \quad [\textsc{s-proxy}]$$

$$\frac{v \text{ is a variable, constant, or address}}{v \sim_G v} \quad [\textsc{s-id}]$$

$$\frac{e \sim_G e'}{\lambda x.\, e \sim_G \lambda x.\, e'} \quad [\textsc{s-fun}]$$

$$\frac{e_1 \sim_G e_1' \qquad e_1 \sim_G e_2'}{e_1\ e_2 \sim_G e_1'\ e_2'} \quad [\textsc{s-call}]$$

$$\frac{e_1 \sim_G e_1' \qquad e_2 \sim_G e_2' \qquad e_3 \sim_G e_3'}{\mathtt{if}\ e_1\ e_2\ e_3 \sim_G \mathtt{if}\ e_1'\ e_2'\ e_3'} \quad [\textsc{s-if}]$$

$$\frac{e \sim_G e'}{uop\ e \sim_G uop\ e'} \quad [\textsc{s-unary}]$$

$$\frac{e_1 \sim_G e_1' \qquad e_2 \sim_G e_2'}{e_1\ bop\ e_2 \sim_G e_1'\ bop\ e_2'} \quad [\textsc{s-binary}]$$

$$\frac{\overline{e_f} \sim_G \overline{e_f'} \qquad \overline{e_v} \sim_G \overline{e_v'}}{\{\, \overline{e_f : e_v}\,\} \sim_G \{\, \overline{e_f' : e_v'}\,\}} \quad [\textsc{s-alloc}]$$

$$\frac{e_1 \sim_G e_1' \qquad e_2 \sim_G e_2'}{e_1[e_2] \sim_G e_1'[e_2']} \quad [\textsc{s-get}]$$

$$\frac{e_1 \sim_G e_1' \qquad e_2 \sim_G e_2' \qquad e_3 \sim_G e_3'}{e_1[e_2] := e_3 \sim_G e_1'[e_2'] := e_3'} \quad [\textsc{s-set}]$$

$$\frac{e_1 \sim_G e_1' \qquad e_2' \sim_G e_2'}{\mathtt{proxy}\ e_1\ e_2 \sim_G \mathtt{proxy}\ e_1'\ e_2'} \quad [\textsc{s-mk-proxy}]$$

domain we have:

$$dom(H_1(a)) = dom(H_2(a))$$
$$\forall\, w \in dom(H_1(a)).\ \ H_1(a)(w)\ \sim_G\ H_2(a)(w)$$

Finally, the simulation relation $(H_1, e_1) \sim (H_2 \uplus G, e_2)$ relates a state (H_1, e_1) with another state $(H_2 \uplus G, e_2)$ with meta-records G, where the heaps and expressions are appropriately simulated with respect to the meta-records:

$$(H_1, e_1) \sim (H_2 \uplus G, e_2) \ \textit{iff}\ e_1\ \sim_G\ e_2 \ \textit{and}\ H_1 \sim_G H_2$$

Here, $H_2 \uplus G$ denotes the union of maps with disjoint domains.

The following lemma shows that this relation $\sim$ is indeed a simulation relation.

LEMMA 1. *Suppose $S_1 \sim S_2$ and $S_1 \rightarrow S_1'$. Then there exists S_2' such that $S_2 \rightarrow^+ S_2'$ and $S_1' \sim S_2'$.*

Proof: We have that

$$S_1 = H_1, e_1 \sim H_2 \uplus G, e_2 = S_2$$

where $H_1 \sim_G H_2$ and $e_1 \sim_G e_2$. The proof proceeds by induction and case analysis on the derivation of $S_1 \rightarrow S_1'$ and by a second induction on the derivation of $e_1 \sim_G e_2$.

- [CALL] In this case

  $$S_1 = H_1, (\lambda x.\, e)\ v \rightarrow H_1, e[x := v] = S_1'$$

  Also, $S_2 = f\ v'$ where $\lambda x.\, e \sim_G f$ and $v \sim_G v'$. This case proceeds by subcase analysis on $\lambda x.\, e \sim_G f$.

  - [S-FUN] For $f = (\lambda x.\, e')$ where $e \sim_G e'$ we have:

    $$\begin{array}{rll} S_2 & = & H_2 \uplus G, (\lambda x.\, e')\ v' \\ & \rightarrow & H_2 \uplus G, e'[x := v'] \quad [\textsc{Call}] \\ & = & S_2' \end{array}$$

    Thus $S_1' \sim S_2'$.

  - [S-PROXY] For $f = (\mathtt{proxy}\ c\ a)$ where $G(a) = \mathtt{HR}_{v''}$ and $(\lambda x.\, e) \sim_G v''$ we have:

    $$\begin{array}{rlll} S_2 & = & H_2 \uplus G, (\mathtt{proxy}\ c\ a)\ v' & \\ & \rightarrow & H_2 \uplus G, a.\mathtt{call}\ v' & [\textsc{CallProxy}] \\ & \rightarrow & H_2 \uplus G, (\lambda y.\, v''\ y)\ v' & [\textsc{Get}] \\ & \rightarrow & H_2 \uplus G, v''\ v' & [\textsc{Call}] \\ & = & S_2'' & \end{array}$$

    Then $(\lambda x.\, e)\ v \sim_G v''\ v'$ by a smaller derivation, so by induction there exists S_2' such that $S_2'' \rightarrow^+ S_2'$ and $S_1' \sim S_2'$.

- [ALLOC] In this case

  $$S_1 = H_1, \{\, \overline{s : v}\,\} \rightarrow H_1[a := \{\, \overline{s : v}\,\}], a = S_1'$$

  where $a \notin dom(H_1)$. Without loss of generality assume $a \notin G$. Then:

  $$\begin{array}{rlll} S_2 & = & H_2 \uplus G, \{\, \overline{s : v'}\,\} & \text{where } \overline{v} \sim_G \overline{v'} \\ & \rightarrow & H_2[a := \{\, \overline{s : v'}\,\}] \uplus G, a & [\textsc{Alloc}] \\ & = & S_2' & \end{array}$$

  Since we have $H_1[a := \{\, \overline{s : v}\,\}] \sim_G H_2[a := \{\, \overline{s : v'}\,\}]$ we also have $S_1' \sim S_2'$.

- [GET] In this case

  $$S_1 \ = \ H_1, a[i] \ \rightarrow \ H_1, v \ = \ S_1'$$

  where $v = H_1(a)(i)$. This case proceeds by subcase analysis on $a[i] \sim_G e_2$.

  - For $e_2 = a[i]$ we have:

    $$\begin{array}{rll} S_2 & = & H_2 \uplus G, a[i] \\ & \rightarrow & H_2 \uplus G, v' \quad [\textsc{Get}] \\ & = & S_2' \end{array}$$

    where $v = H_1(a)(i) \sim_G H_2(a)(i) = v'$.

- For $e_2 = (\texttt{proxy}\ c\ b)[v']$ where $G(b) = \texttt{HR}_w$ and $a \sim_G w$ and $i \sim_G v'$ we have:

$$\begin{array}{rll} S_2 & = & H_2 \uplus G, (\texttt{proxy}\ c\ b)[v'] \\ & \to & H_2 \uplus G, b.\texttt{getr}\ v' \qquad [\text{GetrProxy}] \\ & \to & H_2 \uplus G, (\lambda n.\, w[n])\ v' \qquad [\text{Get}] \\ & \to & H_2 \uplus G, w[v'] \qquad [\text{Call}] \\ & = & S_2'' \end{array}$$

We have $S_1 \sim S_2''$ by a smaller derivation, so by induction there exists S_2' such that $S_2'' \to^+ S_2'$ and $S_1' \sim S_2'$.

- For $e_2 = a[(\texttt{proxy}\ c\ b)]$ where $G(b) = \texttt{HR}_w$ and $i \sim_G w$ we have:

$$\begin{array}{rll} S_2 & = & H_2 \uplus G, a[(\texttt{proxy}\ c\ b)] \\ & \to & H_2 \uplus G, b.\texttt{geti}\ a \qquad [\text{GetiProxy}] \\ & \to & H_2 \uplus G, (\lambda q.\, q[w])\ a \qquad [\text{Get}] \\ & \to & H_2 \uplus G, a[w] \qquad [\text{Call}] \\ & = & S_2'' \end{array}$$

We have $S_1 \sim S_2''$ by a smaller derivation, so by induction there exists S_2' such that $S_2'' \to^+ S_2'$ and $S_1' \sim S_2'$.

- [set] In this case

$$S_1 = H_1, a[i] = v \to H_1', v = S_1'$$

where $H_1' = H_1[a := H_1(a)[i := v]]$. We proceed by subcase analysis on $(a[i] = v) \sim_G e_2$.

- For $e_2 = (a[i] = v')$ where $v \sim_G v'$ we have:

$$\begin{array}{rll} S_2 & = & H_2 \uplus G, a[i] = v' \\ & \to & H_2' \uplus G, v' \qquad [\text{Set}] \\ & = & S_2' \end{array}$$

where $H_2' = H_2[a := H_2(a)[i := v']]$. Since $v \sim_G v'$ we have $H_1' \sim_G H_2'$ and $S_1' \sim S_2'$.

- For $e_2 = ((\texttt{proxy}\ c\ b)[w'] = v')$ where $v \sim_G v'$ and $G(b) = \texttt{HR}_w$ and $a \sim_G w$ and $i \sim_G w'$ we have:

$$\begin{array}{rll} S_2 & = & H_2 \uplus G, (\texttt{proxy}\ c\ b)[w'] = v' \\ & \to & H_2 \uplus G, (b.\texttt{setr}\ w'\ v'); v' \qquad [\text{SetrProxy}] \\ & \to & H_2 \uplus G, (\lambda n, y.\, w[n] = y)\ w'\ v'; v' \qquad [\text{Set}] \\ & \to & H_2 \uplus G, w[w'] = v'; v' \qquad [\text{Call}] \\ & = & S_2'' \end{array}$$

where $H_2' = H_2[a := H_2(a)[w := v']]$. We have $S_1 \sim S_2''$ by a smaller derivation, so by induction there exists S_2' such that $S_2'' \to^+ S_2'$ and $S_1' \sim S_2'$.

- For $e_2 = (a[(\texttt{proxy}\ c\ b)] = v')$ where $G(b) = \texttt{HR}_w$ and $v \sim_G v'$ and $i \sim_G w$ we have:

$$\begin{array}{rll} S_2 & = & H_2 \uplus G, a[\texttt{proxy}\ c\ b] = v' \\ & \to & H_2 \uplus G, (b.\texttt{seti}\ a\ v'); v' \qquad [\text{SetiProxy}] \\ & \to & H_2 \uplus G, (\lambda r, y.\, r[w] = y)\ a\ v'; v' \qquad [\text{Set}] \\ & \to & H_2 \uplus G, a[w] := v'; v' \qquad [\text{Call}] \\ & = & S_2'' \end{array}$$

where $H_2' = H_2[a := H_2(a)[w := v']]$. We have $S_1 \sim S_2''$ by a smaller derivation, so by induction there exists S_2' such that $S_2'' \to^+ S_2'$ and $S_1' \sim S_2'$.

- [s-unary] In this case

$$S_1 = H_1, \mathit{uop}\ r \to H_1, \delta(\mathit{uop}, r) = S_1'$$

This case proceeds by subcase analysis on $\mathit{uop}\ r \sim_G e_2$.

- For $e_2 = \mathit{uop}\ r'$ where $r \sim_G r'$ we have:

$$\begin{array}{rll} S_2 & = & H_2 \uplus G, \mathit{uop}\ r' \\ & \to & H_2 \uplus G, \delta(\mathit{uop}, r') \qquad [\text{UnaryOp}] \\ & = & S_2' \end{array}$$

Thus $S_1' \sim S_2'$.

- For $e_2 = \mathit{uop}\ (\texttt{proxy}\ c\ a)$ where $G(a) = \texttt{HR}_v, r \sim_g v$ we have:

$$\begin{array}{rll} S_2 & = & H_2 \uplus G, \mathit{uop}\ (\texttt{proxy}\ c\ a) \\ & \to & H_2 \uplus G, a.\texttt{unary}\ \texttt{"}\mathit{uop}\texttt{"} \qquad [\text{UnaryProxy}] \\ & \to^+ & H_2 \uplus G', \mathit{uop}\ v \qquad G' \supset G \\ & = & S_2'' \end{array}$$

We have $S_1 \sim S_2''$ by a smaller derivation, so by induction there exists S_2' such that $S_2'' \to^+ S_2'$ and $S_1' \sim S_2'$.

- [s-binary] In this case

$$S_1 = H_1, r_1\ \mathit{bop}\ r_2 \to H_1, \delta(\mathit{bop}, r_1, r_2) = S_1'$$

This case proceeds by a subcase analysis on the derivation of $(r_1\ \mathit{bop}\ r_2) \sim_G e_2$.

- For $e_2 = (r_1'\ \mathit{bop}\ r_2')$ where $r_1 \sim_G r_1'$ and $r_2 \sim_G r_2'$ we have:

$$\begin{array}{rll} S_2 & = & H_2 \uplus G, r_1'\ \mathit{bop}\ r_2' \\ & \to & H_2 \uplus G, \delta(\mathit{bop}, r_1', r_2') \qquad [\text{BinaryOp}] \\ & = & S_2' \end{array}$$

Thus $S_1' \sim S_2'$.

- For $e_2 = ((\texttt{proxy}\ c\ a)\ \mathit{bop}\ r_2')$ where $G(a) = \texttt{HR}_v$ and $r_1 \sim_G v$ and $r_2 \sim_G r_2'$ we have:

$$\begin{array}{rll} S_2 & = & H_2 \uplus G, (\texttt{proxy}\ c\ a)\ \mathit{bop}\ r_2' \\ & \to & H_2 \uplus G, a.\texttt{left}\ r_2' \qquad [\text{LeftProxy}] \\ & \to^+ & H_2 \uplus G', v\ \mathit{bop}\ r_2' \qquad G' \supset G \\ & = & S_2'' \end{array}$$

We have $S_1 \sim S_2''$ by a smaller derivation, so by induction there exists S_2' such that $S_2'' \to^+ S_2'$ and $S_1' \sim S_2'$.

- For $e_2 = r_1'\ \mathit{bop}\ (\texttt{proxy}\ c\ a)$ where $G(a) = \texttt{HR}_v$ and $r_1 \sim_G r_1'$ and $r_2 \sim_G v$ we have:

$$\begin{array}{rll} S_2 & = & H_2 \uplus G, r_1'\ \mathit{bop}\ (\texttt{proxy}\ c\ a) \\ & \to & H_2 \uplus G, a.\texttt{right}\ r_1' \qquad [\text{LeftProxy}] \\ & \to^+ & H_2 \uplus G', r_1'\ \mathit{bop}\ v \qquad G' \supset G \\ & = & S_2'' \end{array}$$

We have $S_1 \sim S_2''$ by a smaller derivation, so by induction there exists S_2' such that $S_2'' \rightarrow^+ S_2'$ and $S_1' \sim S_2'$.

- [IfTrue] In this case

$$S_1 = H_1, \texttt{if}\ r\ e_1\ e_2 \rightarrow H_1, e_1 = S_1'$$

where $r = \texttt{true}$. This case proceeds by subcase analysis on $(\texttt{if}\ r\ e_1\ e_2) \sim_G e_3$.

  - For $e_3 = (\texttt{if}\ r\ e_1'\ e_2')$ where $e_1 \sim_G e_1'$ and $e_2 \sim_G e_2'$ we have:

$$\begin{array}{rcll} S_2 & = & H_2 \uplus G, \texttt{if}\ r\ e_1'\ e_2' & \\ & \rightarrow & H_2 \uplus G, e_1' & [\textsc{IfTrue}] \\ & = & S_2' & \end{array}$$

    Thus $S_1' \sim S_2'$.

  - For $e_3 = (\texttt{if}\ (\texttt{proxy}\ c\ a)\ e_1'\ e_2')$ where $G(a) = \texttt{HR}_v$ and $r \sim_G v$ and $e_1 \sim_G e_1'$ and $e_2 \sim_G e_2'$ we have:

$$\begin{array}{rcll} S_2 & = & H_2 \uplus G, \texttt{if}\ (\texttt{proxy}\ c\ a)\ e_1'\ e_2' & \\ & \rightarrow & H_2 \uplus G, \texttt{if}\ (a.\texttt{test}())\ e_1'\ e_2' & [\textsc{TestProxy}] \\ & \rightarrow^+ & H_2 \uplus G, \texttt{if}\ v\ e_1'\ e_2' & [\textsc{Get}], [\textsc{Call}] \\ & = & S_2'' & \end{array}$$

    By induction there exists S_2' such that $S_2'' \rightarrow^+ S_2'$ and $S_1' \sim S_2'$.

- [IfFalse] Similar to the previous case.
- [IsProxy],[NotProxy],[UnProxy],[UnProxyFalse] The simulation relation does not permit the `isProxy` or `unProxy` primitives in e_1.
- [context] In this case

$$S_1 = H_1, E_1[e_1] \rightarrow H_1', E_1[e_1'] = S_1'$$

where $H_1, e_1 \rightarrow H_1', e_1'$. We extend the $\sim_G$ relation from expressions to evaluation contexts in a compatible manner. In order to have $S_1 \sim S_2$ we must have $S_2 = H_2 \uplus G, e_3$ where $H_1 \sim_G H_2$ and $E_1[e_1] \sim_G e_3$. So $\exists E_2, e_2$ such that $E_2[e_2] = e_3$ and $E_1 \sim_G E_2$ and $e_1 \sim_G e_2$. Since

$$\begin{array}{rcl} H_1, e_1 & \sim & H_2 \uplus G, e_2 \\ H_1, e_1 & \rightarrow & H_1', e_1' \end{array}$$

by induction there exists H_2', e_2' such that

$$\begin{array}{rcl} H_2 \uplus G, e_2 & \rightarrow & H_2' \uplus G, e_2' \\ H_1', e_1' & \sim & H_2' \uplus G, e_2' \end{array}$$

By the [Context] rule we have

$$S_2 = H_2 \uplus G, E_2[e_2] \rightarrow H_2' \uplus G, E_2[e_2'] = S_2'$$

and $S_1' \sim S_2'$.