Virtual Values for Language Extension

Thomas H. Austin, Tim Disney, and 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; units of measure; and symbolic execution. We report on our experience implementing virtual values for Javascript within an extension for the Firefox browser.

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 and successful 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 entire 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. We formalize the semantics of virtual values in the context of a particular dynamically typed language; however, our ideas should be generally extensible to other languages, particularly to languages that already perform dynamic type checks. The operational semantics of our language is straightforward, with additional evaluation rules for invoking traps for operations on virtual values.

We believe that virtual values provide a rather useful and powerful degree of language extensibility. Of course, validating a language design feature is always difficult. 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 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. Symbolic execution, which increases test coverage by executing code on symbolic input values [20, 13].
6. Dynamically checked contracts [8], 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, after which further interaction is forbidden [22].

Each language extension is powerful yet small (the complete code is included in this paper), 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 contracts, and use that contract extension to document other extensions.

[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.

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 may cross-cut the other features and evaluation rules of the language. For example, the information flow and complex number extension would 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 coherent and extensible architecture. Composed virtual values are essentially an instance of the Decorator Pattern [12]. This pattern 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* [23, 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* [10].

Virtual values generalizes these prior ideas to virtualize the interface between code and *all data values*. This generalization provides significant benefits, and it enables additional interesting applications, most notably (1)–(5) from the list above.

SmallTalk [14] 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 significant flexibility, and in some sense already virtualizes the interface between code and data, since all operations are performed via dynamically-dispatched method calls. Indeed, many of the extensions that we propose could also be implemented in Smalltalk, or in other pure object languages such as AmbientTalk [25], E [24], or Python. A central contribution of this paper is to demonstrate that this degree of extensibility is not restricted to pure object languages; virtual values enable similar extensibility in languages that include non-object values, and in languages that are not object oriented.

From another perspective, virtual values can be considered a type of object, since they carry their own behavior. Even though there is no `this` binding. In this sense, virtual values provide a means to enrich a non-object language (or a language with non-object values) with the extensibility benefits of pure object

languages, since all operations can be dynamically dispatched via virtual value traps.

Language extensibility has been the target of a rich body of prior research. For example, CLOS provides a very flexible *metaobject protocol* (MOP) [19], 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) [18] 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]. 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 [27] type classes. This static type based approach provides stronger correctness guarantees and improved performance over virtual values, but at a 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 with non-object values (or non object-oriented languages) can still enjoy the extensibility benefits of pure object languages;
- it presents a formal yet accessible operational semantics for virtual values;
- it demonstrates the extensibility benefits of virtual values by implementing seven powerful language extensions: (1) complex numbers; (2) units of measure; (3) delayed evaluation; (4) taint analysis; (5) symbolic execution; (6) contracts; and (7) 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}.

Figure 1: $\lambda_{\mathbf{proxy}}$ Syntax

$$
\begin{array}{lll}
e ::= x \mid c \mid \lambda x.\, e \mid e\ e \mid \texttt{if}\ e\ e\ e \mid uop\ e \mid e\ bop\ e \mid \{\,\overline{e : e}\,\} & & \textbf{Expressions} \\
\quad e[e] \mid e[e] := e \mid \texttt{proxy}\ e \mid \texttt{isProxy}\ e & & \\
c \quad ::= n \mid s \mid \texttt{false} \mid \texttt{true} \mid \texttt{unit} & & \textbf{Constants} \\
uop ::= - \mid ! \mid \texttt{isNum} \mid \texttt{isBool} \mid \texttt{isFunction} \mid \texttt{isRecord} \mid \texttt{tostring} \mid \ldots & & \textbf{Unary operators} \\
bop ::= + \mid = \mid != \mid \ldots & & \textbf{Binary operators}
\end{array}
$$

Syntactic Sugar

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

$$
\begin{array}{rcl}
\texttt{letrec}\ x = e_1;\ e_2 & \stackrel{\text{def}}{=} & \texttt{let}\ y = \{\};\ y.x := \theta e_1;\ \theta e_2 \quad \text{where } \theta = [x := y.x] \\
\overline{\texttt{private}\ x = e};\ \overline{y = e'} & \stackrel{\text{def}}{=} & \texttt{let}\ p = \{\};\ \texttt{let}\ q = \{\};\ \ \overline{p.x := \theta e};\ \ \overline{q.y := \theta e'};\ \ q \\
 & & \text{where } \theta = [\overline{x := p.x}, \overline{y := q.y}]
\end{array}
$$

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 ($uop\ e$ 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 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 the following nine *trap functions*:

$$
\begin{array}{ll}
\texttt{call} :: \texttt{argument} \to \texttt{result} & \texttt{unary} :: uop \to \texttt{result} \\
\texttt{getr} :: \texttt{index} \to \texttt{contents} & \texttt{left} :: bop \to \texttt{rightarg} \to \texttt{result} \\
\texttt{geti} :: \texttt{record} \to \texttt{contents} & \texttt{right} :: bop \to \texttt{leftarg} \to \texttt{result} \\
\texttt{setr} :: \texttt{index} \to \texttt{newcontents} \to \texttt{Unit} & \texttt{test} :: \texttt{Unit} \to \texttt{Any} \\
\texttt{seti} :: \texttt{record} \to \texttt{newcontents} \to \texttt{Unit} &
\end{array}
$$

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 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 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 record *index* i is a constant or an address. A *raw value* r is a constant, an address, or a λ-expression. A *value* v is either a raw value or else `proxy` v, where v 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 excludes comparing λ-expressions.

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

The 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 (`proxy` 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` 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].

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

Runtime Syntax:

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

Evaluation Rules:

$$
\begin{array}{rll}
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[i] \rightarrow H, v & i \in \mathit{dom}(H(a)), v = H(a)(i) & [\textsc{Get}] \\
H, a[i] \rightarrow H, \texttt{false} & i \notin \mathit{dom}(H(a)) & [\textsc{GetFalse}] \\
H, a[i] := v \rightarrow H', v & H' = H[a := H(a)[i := 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 false}\ e_1\ e_2 \rightarrow H, e_2 & & [\textsc{IfFalse}] \\
H, \texttt{isProxy}\ (\texttt{proxy}\ h) \rightarrow H, \texttt{true} & & [\textsc{IsProxy}] \\
H, \texttt{isProxy}\ r \rightarrow H, \texttt{false} & & [\textsc{IsNotProxy}] \\
 & & \\
H, (\texttt{proxy}\ h)\ v \rightarrow H, h.\texttt{call}\ v & & [\textsc{CallProxy}] \\
H, (\texttt{proxy}\ h)[w] \rightarrow H, h.\texttt{getr}\ w & & [\textsc{GetrProxy}] \\
H, r[\texttt{proxy}\ h] \rightarrow H, h.\texttt{geti}\ r & & [\textsc{GetiProxy}] \\
H, (\texttt{proxy}\ h)[w] := v \rightarrow H, (h.\texttt{setr}\ w\ v);\ v & & [\textsc{SetrProxy}] \\
H, r[\texttt{proxy}\ h] := v \rightarrow H, (h.\texttt{seti}\ r\ v);\ v & & [\textsc{SetiProxy}] \\
H, \mathit{uop}\ (\texttt{proxy}\ h) \rightarrow H, h.\texttt{unary}\ \texttt{"}\mathit{uop}\texttt{"} & & [\textsc{UnaryProxy}] \\
H, (\texttt{proxy}\ h)\ \mathit{bop}\ v \rightarrow H, h.\texttt{left}\ \texttt{"}\mathit{bop}\texttt{"}\ v & & [\textsc{LeftProxy}] \\
H, r\ \mathit{bop}\ (\texttt{proxy}\ h) \rightarrow H, h.\texttt{right}\ \texttt{"}\mathit{bop}\texttt{"}\ r & & [\textsc{RightProxy}] \\
H, \texttt{if}\ (\texttt{proxy}\ 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}
$$

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, 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 do not provide an `isProxy` trap, and so the `isProxy` construct allows value consumers to reliably identify proxies and so to defend against unwanted proxy behaviors.

2.4 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, 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 the proxies provided by λ_{proxy}, we next present a series of progressively more 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 traps simply get stuck if their proxy is used inappropriately. For documentation purposes, each definition includes a *contract*, whose semantics we formalize (via proxies) in section 5 below.

```
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
}
```

Fig. 3. Identity Proxy Extension

```
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()
    }
```

Fig. 4. Lazy Evaluation Extension

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 `binaryOps` lookup table.

The `identityProxy` may appear to be circular, since it defines each unary operation in terms of that operation itself. To illustrate how this circularity bottoms out, consider the evaluation of −(`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 entirely 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 this proxy is indeed semantically transparent. Achieving this degree of transparency required several careful design choices in our language semantics–for example, the equality operator cannot distinguish λ-expressions.

4 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 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.

5 Contract Extension

A *contract* [8] 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.

A flat contract has the form $\texttt{(Flat}_c\ \texttt{pred)}$. When applied to an argument `x`, this contract requires that `x` satisfy the predicate `pred`. A function contract $\texttt{(Function}_c\ \texttt{Domain Range)}$ 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 $\texttt{(Map}_c\ \texttt{Domain Range)}$; 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 has the form $\texttt{(Record}_c\ \texttt{contracts)}$, 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]`. Both kinds of record contracts are enforced in a lazy manner, on each access and update of the resulting proxy.

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

```
// 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)
```

Fig. 5. Contracts Extension

```
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))
```

Fig. 6. Tainting Extension

We use the syntax `Domain → Range` and `Domain ⇒ Range` to abbreviate function and map contracts, respectively, and {{ $\overline{s : \texttt{Contract}}$ }} for heterogeneous record contracts. 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.

$$
\begin{array}{rcl}
\texttt{Domain -> Range} & \stackrel{\text{def}}{=} & \texttt{Function}_c\ \texttt{Domain Range} \\
\texttt{Domain => Range} & \stackrel{\text{def}}{=} & \texttt{Map}_c\ \texttt{Domain Range} \\
\{\!\{\ \overline{s : \text{Contract}}\ \}\!\} & \stackrel{\text{def}}{=} & \texttt{Record}_c\ \{\ \overline{s : \text{Contract}}\ \} \\
\overline{\texttt{private}\ x :: C = e};\ \overline{y :: C' = e'} & \stackrel{\text{def}}{=} & \texttt{let}\ p = \{\};\ \texttt{let}\ q = \{\}; \\
 & & \quad \overline{p.x := \theta(C\ e)};\ \overline{q.y := \theta(C'\ e')};\ q \\
 & & (\text{where } \theta = [\overline{x := p.x}, \overline{y := q.y}])
\end{array}
$$

6 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 runtime data representations.

Proxies allow this complexity to be isolated into a small extension module, as shown in figure 6. 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` but then taint the result.

To untaint values (after they have been sanitized) 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.

The `unproxy` table may raise some concerns about potential memory leaks, if `unproxy[p]` remains live even after `p` has been collected. Although the semantics of λ_{proxy} does not formalize garbage collection behavior, λ_{proxy} records could be implemented as *ephemeron tables* [16], where the entry for `unproxy[p]` is collected as soon as `p` is garbage, thus alleviating these concerns.

7 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 underlying language implementation, and to manipulate these additional types using traditional and intuitive operator syntax. In many languages, this kind of extension is difficult. For example, Java does provide `Bignum`s, but only as a library with awkward method invocation syntax, and it does not provide rational numbers, complex numbers, or decimal floating point numbers.

Figure 7 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. For binary operations, the `left` trap first converts the right argument `y` (which should be a ordinary number or a complex number) into a (real, imaginary) pair, 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.

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

```
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
    // all Complex are non-false
    test : λ.    true
  }
  unproxy[p] := pair
  p

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

i :: Complex = makeComplex 0 1

Complex = Flat_c isComplex
```

Fig. 7. Complex Number Extension

```
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)
```

Fig. 8. Revokable Membrane Extn.

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

8 Revokable Membranes

Figure 8 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).

Figure 8 illustrates how to implement revocable membranes via proxies. 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`], which maps from references known to the `true` component to corresponding references in the `false` component; and `unproxy`[`false`], which is the 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 contain object references. 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.

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 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, and so both language extensions compose nicely.

9 Dynamic Units Of Measure

Several type systems (see for example, [17]) 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.

Proxies provide a convenient means to track units dynamically, and we provide an example in figure 10. The contract `UNum` describes a number, possibly wrapped in a chain of proxies, each of which 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 be used by client code to create desired units of measure, as in:

```
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)
```

Fig. 9. Symbolic Execution Extension

```
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 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
       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 ⇒ String → 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))
  ···
}

// left arg never a proxy
private unitRightOps :: BinaryOp ⇒
String → Int → UNum → Any → Any
= {
  "*": λu,i,n,l. makeUNum u i (l * n)
  ···
}

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)
```

Fig. 10. Units of Measure Extension

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

10 Symbolic Execution

Achieving good test coverage using traditional testing is notoriously difficult, since it requires first pre-commiting to specific test inputs, and hoping that those inputs then drive execution through appropriate control-flow paths.

Symbolic execution provides a method for achieving greater test coverage by exploring the behavior of the target program on an initially undetermined *symbolic* input [20, 13], and incrementally refining or constraining that input as each successive control-flow branch in the target program is encountered. Typically, symbolic execution is performed via a specialized interpreter, or by appropriately instrumenting the program source or bytecode.

Figure 9 illustrates a lightweight proxy-based approach that extends a standard execution engine (interpreter/compiler/JIT) to perform symbolic execution, simply by designing appropriate *symbolic proxies*.

The first nine lines of this figure describe an API symbolic reasoning, which could be implemented on top of a standard SMT solver such as Simplify [6] or Z3 [5]. A *symbolic expression* is essentially a tree with unary and binary operators on internal nodes (created via `SE_unary` and `SE_binary`) and with symbolic variables and constants at the leafs (created via `SE_var` and `SE_constant`). The API also maintains a collection of constraints, where (`SE_constrain se`) adds an additional constraint that the symbolic expression `se` must hold. Finally, the function call (`SE_sat se`) checks if the symbolic expression `se` is satisfiable given the current constraints (that is, it checks if the current constraint set plus the additional constraint `se` is satisfiable).

Using this API, we generate *symbolic proxies* that can be passed to the software under test. The variable `unproxy` maps these 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 proxy `p`, where the `unary`, `left`, and `right` traps of `p` perform the appropriate API operations on `se` to yield a new symbolic expression that is then wrapped in a proxy. Thus, for example, if X is an `SE_var`, then the expression (`1 + (2 * toSymProxy(X))`) evaluates to a proxy containing a symbolic representation of "`1 + (2 * X)`". Essentially, the operator traps simply record how new symbolic values are generated from previous ones.

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 explore a

branch not yet tested, in order to maximize branch or path coverage). Once the branch `choice` is chosen, the trap handler records the new constraint and then returns `choice` to the [TestProxy] rule.

The symbolic execution module 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 demonstrates the key ideas.

As an example, suppose we wanted to test a function `sort` that takes as input an array, using a function `checkSorted` to check the result 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

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

Unfortunately, this test suite is rather ill-chosen, since both test inputs will execute the same code path through `sort` (assuming the `sort` is implemented by repeated comparisons, rather than by *e.g.* bucket sorting). 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:

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

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

As expected, symbolic proxies are compatible with other proxy extensions. For example, we can generate symbolic complex numbers by evaluating:

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

11 Implementation in 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 [26] for Firefox, which is based on the Narcissus [30] 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

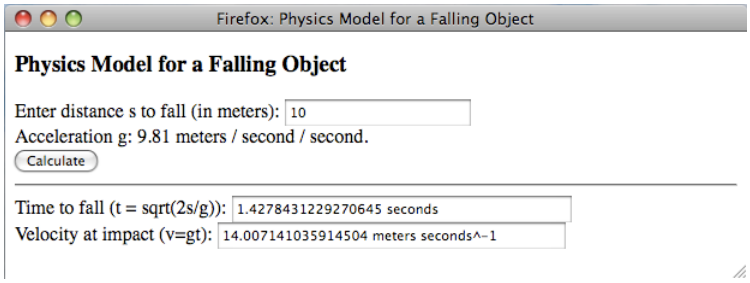

Fig. 11. Sample web page with a Javascript evaluator running in Firefox

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 brief web application that uses the units of measurement proxy 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 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. 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 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 [11].

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 v" could represent a proxy for the value v, where the handler a 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 three words: a header word plus slots for a and v. Overall, it appears likely that proxies can be implemented fairly efficiently, particular in a dynamic languages.

```
1 if (JSVAL_IS_INT(rval)&&(i=JSVAL_TO_INT(rval))!=0) {
2     i = -i; regs.sp[-1] = INT_TO_JSVAL(i);     // Fast path for ints
3 } else if (JSVAL_IS_DOUBLE(rval)) {
4     ···       // Second fast path for doubles
5 } else {
6     ··· // Slow path for handling implicit conversions
7     ··· // Ideal spot for handling proxies
8     ··· // Error handling
9 }
```

Fig. 12. Tag Checks in SpiderMonkey's Unary Minus Code

12 Discussion and Future Work

The language extensions presented in Sections 3–10 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 intuitive and convenient.

The introduction of virtual values does significantly change the semantics and denotational structure of the language, and suggests that further study of the resulting denotational semantics is required. In particular, a full abstraction result [3] for λ_{proxy} 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 still 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 [8, 7], language interoperation via proxies [15], and hybrid and gradual typing [28, 9], and space-efficient gradual typing [29]. We conjecture that 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.

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. 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. 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. R. B. Findler and M. Blume. Contracts as pairs of projections. In *International Symposium on Functional and Logic Programming*, pages 226–241, 2006.
8. R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *Proceedings of the International Conference on Functional Programming*, pages 48–59, 2002.

9. C. Flanagan. Hybrid type checking. In *Symposium on Principles of Programming Languages*, pages 245 – 256, 2006.
10. M. Flatt and PLT. Reference: Racket. Technical Report PLT-TR2010-1, PLT Inc., June 7, 2010. `http://racket-lang.org/tr1/`.
11. 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.
12. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, Boston, MA, 1995.
13. P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Conference on Programming Language Design and Implementation*, pages 213–223, 2005.
14. A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.
15. K. E. Gray, R. B. Findler, and M. Flatt. Fine-grained interoperability through mirrors and contracts. In *OOPSLA*, pages 231–245, 2005.
16. B. Hayes. Ephemerons: a new finalization mechanism. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 176–183, New York, NY, USA, 1997. ACM.
17. A. Kennedy. Relational parametricity and units of measure. In *Principles of Programming Languages*, pages 442–455, 1997.
18. G. Kiczales. Aspect-oriented programming. *ACM Comput. Surv.*, page 154, 1996.
19. G. Kiczales, J. D. Rivieres, and D. G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, July 1991.
20. J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.
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. Mozilla labs: Zaphod addon. `http://mozillalabs.com/zaphod`, accessed October 2010.
27. 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.
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. Wikipedia: Narcissus JavaScript engine. `http://en.wikipedia.org/wiki/Narcissus_%28JavaScript_engine%29`, accessed October 2010.

Figure 13: Revised Identity 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.   { "-"    : 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
}
```

A Correctness Proof for the Identity Proxy

In this appendix, we provide an illustrative proof that the identity proxy is semantically transparent. Correctness proofs for the other language extensions are similar but somewhat more involved.

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 v.

As defined earlier, a state $S = H, e$ is a pair of a heap and an expression. We say a closed expression e *converges*, written $e\downarrow$, if $\emptyset, e \rightarrow^* S$ for some irreducible state S. Two expressions are *contextually equivalent*, written $e_1 \cong e_2$, if for all contexts C,

$$C[e_1]\downarrow \Leftrightarrow C[e_2]\downarrow$$

To show that a value v and its corresponding proxy $\texttt{HR}_v$ are contextually equivalently, we define a simulation relation that relates the evaluations of $C[v]$ and $C[\texttt{HR}_v]$. The evaluation of $C[\texttt{HR}_v]$ will execute additional proxy code that will allocate additional records in the heap. We refer to these proxy-allocated records as *meta*-records, and use G to refer to the meta portion of the heap. We define the simulation relation $\sim$ on evaluation states as:

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

Figure 14: Definition of $\sim_G$

$$\frac{}{e \sim_G e} \quad \text{[IDENTITY]}$$

$$\frac{G(a) = \mathtt{HR}_{v'} \quad v \sim_G v'}{v \sim_G \mathtt{proxy}\ a} \quad \text{[PROXY]}$$

$$\frac{e \sim_G e'}{\lambda x.\, e \sim_G \lambda x.\, e'} \quad \text{[ABSTRACTION]}$$

$$\frac{e_1 \sim_G e_1' \quad e_1 \sim_G e_2'}{e_1\ e_2 \sim_G e_1'\ e_2'} \quad \text{[APPLICATION]}$$

$$\frac{e_1 \sim_G e_1' \quad e_2 \sim_G e_2' \quad e_3 \sim_G e_3'}{\mathtt{if}\ e_1\ e_2\ e_3 \sim_G \mathtt{if}\ e_1'\ e_2'\ e_3'} \quad \text{[CONDITIONAL]}$$

$$\frac{e \sim_G e'}{uop\ e \sim_G uop\ e'} \quad \text{[UNARY]}$$

$$\frac{e_1 \sim_G e_1' \quad e_2 \sim_G e_2'}{e_1\ bop\ e_2 \sim_G e_1'\ bop\ e_2'} \quad \text{[BINARY]}$$

$$\frac{\overline{e_f} \sim_G \overline{e_f'} \quad \overline{e_v} \sim_G \overline{e_v'}}{\{\, \overline{e_f : e_v} \,\} \sim_G \{\, \overline{e_f' : e_v'} \,\}} \quad \text{[RECORD CREATION]}$$

$$\frac{e_1 \sim_G e_1' \quad e_2 \sim_G e_2'}{e_1[e_2] \sim_G e_1'[e_2']} \quad \text{[RECORD LOOKUP]}$$

$$\frac{e_1 \sim_G e_1' \quad e_2 \sim_G e_2' \quad e_3 \sim_G e_3'}{e_1[e_2] := e_3 \sim_G e_1'[e_2'] := e_3'} \quad \text{[RECORD UPDATE]}$$

$$\frac{e \sim_G e'}{\mathtt{proxy}\ e \sim_G \mathtt{proxy}\ e'} \quad \text{[PROXY CREATION]}$$

$$\frac{e \sim_G e'}{\mathtt{isProxy}\ e \sim_G \mathtt{isProxy}\ e'} \quad \text{[PROXY PREDICATE]}$$

where figure 14 defines the simulation relation $e_1 \sim_G e_2$ on expressions with respect to a meta-heap G, and we define $H_1 \sim_G H_2$ to hold if:

$$\begin{array}{l} dom(H_1) = dom(H_2) \\ \mathtt{and}\ \forall\ a \in dom(H_1).\ dom(H_1(a)) = dom(H_2(a)) \\ \mathtt{and}\ \forall\ w \in dom(H_1(a)).\ H_1(a)(w)\ \sim_G\ H_2(a)(w) \end{array}$$

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

Lemma 1. *Suppose* $S_1 \sim S_2$ *and* $S_1 \to S_1'$. *Then there exists* S_2' *st.* $S_2 \to^+ S_2'$ *and* $S_1' \sim S_2'$.

Proof: We have that

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

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

- [CALL] In this case

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

  Also, $S_2 = f\ v'$ where $\lambda x.\, e \sim_G f$ and $v \sim_G v'$. This case proceeds by subcase analysis on $\lambda x.\, e \sim_G f$.
  - For $f = (\lambda x.\, e')$ where $e \sim_G e'$ we have:

$$\begin{array}{rl} S_2 & = H_2 + G, (\lambda x.\, e')\ v' \\ & \to H_2 + G, e'[x := v'] \text{ [CALL]} \\ & = S_2' \end{array}$$

    Thus we have $S_1' \sim S_2'$.
  - For $f = (\texttt{proxy}\ a)$ where $G(a) = \texttt{HR}_{v''}$ and $(\lambda x.\, e) \sim_G v''$ we have:

$$\begin{array}{rll} S_2 & = H_2 + G, (\texttt{proxy}\ a)\ v' & \\ & \to H_2 + G, a.\texttt{call}\ v' & \text{[CALLPROXY]} \\ & \to H_2 + G, (\lambda y.\, v''\ y)\ v' & \text{[CALL]} \\ & \to H_2 + G, v''\ v' & \text{[CALL]} \\ & = S_2'' & \end{array}$$

    Then $(\lambda x.\, e)\ v \sim_G v''\ v'$ by a smaller derivation, so by induction $\exists\ S_2'\ st.\ S_2'' \to^+ S_2'$ and $S_1' \sim S_2'$.
- [ALLOC] In this case

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

  where $a \notin dom(H_1)$. Since e_2 has a single case we have:

$$\begin{array}{rl} S_2 & = H_2 + G, \{\ \overline{s' : v'}\ \} \\ & \to H_2[a := \{\ \overline{s' : v'}\ \}] + G, a \text{ [ALLOC]} \\ & = S_2' \end{array}$$

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

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

  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:

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

  where $v = H_1(a)(i) \sim_G H_2(a)(r) = v'$.
- For $e_2 = (\texttt{proxy}\ b)[v']$ where $G(b) = \texttt{HR}_v$ and $a \sim_G v$ and $i \sim_G v'$ we have:

$$\begin{array}{rll} S_2 &= H_2 + G, (\texttt{proxy}\ b)[v'] & \\ &\to H_2 + G, b.\texttt{getr}\ v' & [\textsc{GetrProxy}] \\ &\to H_2 + G, (\lambda n.\, v[n])\ v' & [\textsc{Getr}] \\ &\to H_2 + G, v[v'] & [\textsc{Call}] \\ &= S_2'' & \end{array}$$

  By induction $\exists\ S_2'\ st.\ S_2'' \to^+ S_2'$ and $S_1' \sim S_2'$.
- For $e_2 = a[(\texttt{proxy}\ b)]$ where $G(b) = \texttt{HR}_v$ and $i \sim_G v$ we have:

$$\begin{array}{rll} S_2 &= H_2 + G, a[(\texttt{proxy}\ b)] & \\ &\to H_2 + G, b.\texttt{geti}\ a & [\textsc{GetiProxy}] \\ &\to H_2 + G, (\lambda q.\, q[v])\ a & [\textsc{Geti}] \\ &\to H_2 + G, a[v] & [\textsc{Call}] \\ &= S_2'' & \end{array}$$

  We have $S_1 \sim S_2''$ by a smaller derivation, so by induction $\exists\ S_2'\ st.\ S_2'' \to^+ S_2'$ 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:

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

  where $H_2' = H_2[a := H_2(a)[i := v']]$. Since we know $v \sim_G v'$ we have $H_1' \sim_G H_2'$ and $S_1' \sim S_2'$.
- For $e_2 = (\texttt{proxy}\ b)[w'] = v'$ where $v \sim_G v', G(b) = \texttt{HR}_w, a \sim_G w$ and $i \sim_G w'$ we have:

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

  where $H_2' = H_2[a := H_2(a)[w := v']]$. By induction $\exists\ S_2'\ st.\ S_2'' \to^+ S_2'$ and $S_1' \sim S_2'$.

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

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

  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 $\exists\ S_2'\ st.\ S_2'' \to^+ S_2'$ and $S_1' \sim S_2'$.

– [Unary] In this case

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

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

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

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

  Thus we have $S_1' \sim S_2'$.
- For $e_2 = uop\ (\texttt{proxy}\ a)$ where $G(a) = \texttt{HR}_v, r \sim_g v$ we have:

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

  By induction $\exists\ S_2'\ st.\ S_2'' \to^+ S_2'$ and $S_1' \sim S_2'$.

– [Binary] In this case

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

This case proceeds by subcase analysis on $r_1\ bop\ r_2 \sim_G e_2$.

- For $e_2 = r_1'\ bop\ r_2'$ where $r_1 \sim_G r_1', r_2 \sim_G r_2'$ we have:

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

  Thus we have $S_1' \sim S_2'$.
- For $e_2 = (\texttt{proxy}\ a)\ bopr_2'$ where $G(a) = \texttt{HR}_v, r_1 \sim_G v, r_2 \sim_G r_2'$ we have:

$$\begin{array}{rll} S_2 &= H_2 + G, (\texttt{proxy}\ a)\ bop\ r_2' & \\ &\to H_2 + G, a.\texttt{left}\ r_2' & [\textsc{LeftProxy}] \\ &\to H_2 + G, v\ bop\ r_2' & [\textsc{Left}] \\ &= S_2'' & \end{array}$$

  By induction $\exists\ S_2'\ st.\ S_2'' \to^+ S_2'$ and $S_1' \sim S_2'$.

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

$$\begin{array}{rll} S_2 &= H_2 + G, r_1'\ bop\ (\texttt{proxy}\ a) & \\ &\rightarrow H_2 + G, a.\texttt{right}\ r_1' & [\textsc{LeftProxy}] \\ &\rightarrow H_2 + G, r_1'\ bop\ v & [\textsc{Right}] \\ &= S_2'' & \end{array}$$

  By induction $\exists\ S_2'\ st.\ S_2'' \rightarrow^+ S_2'$ and $S_1' \sim S_2'$.

– [IfTrue] In this case

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

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 $e_1 \sim_G e_1', e_2 \sim_G e_2'$ we have:

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

  Thus we have $S_1' \sim S_2'$.

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

$$\begin{array}{rll} S_2 &= H_2 + G, \texttt{if}\ (\texttt{proxy}\ a)\ e_1'\ e_2' & \\ &\rightarrow H_2 + G, \texttt{if}\ (a.\texttt{test}())\ e_1'\ e_2' & [\textsc{TestProxy}] \\ &\rightarrow H_2 + G, \texttt{if}\ v\ e_1'\ e_2' & [\textsc{Test}] \\ &= S_2'' & \end{array}$$

  By induction $\exists\ S_2'\ st.\ S_2'' \rightarrow^+ S_2'$ and $S_1' \sim S_2'$.

– [IfFalse] In this case

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

This case proceeds by subcase analysis on $\texttt{if false}\ e_1\ e_2 \sim_G e_3$.

- For $e_3 = \texttt{if false}\ e_1'\ e_2'$ where $e_1 \sim_G e_1', e_2 \sim_G e_2'$ we have:

$$\begin{array}{rll} S_2 &= H_2 + G, \texttt{if false}\ e_1'\ e_2' & \\ &\rightarrow H_2 + G, e_2' & [\textsc{IfFalse}] \\ &= S_2' & \end{array}$$

  Thus we have $S_1' \sim S_2'$.

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

$$\begin{array}{rll} S_2 &= H_2 + G, \texttt{if}\ (\texttt{proxy}\ a)\ e_1'\ e_2' & \\ &\rightarrow H_2 + G, \texttt{if}\ (a.\texttt{test}())\ e_1'\ e_2' & [\textsc{TestProxy}] \\ &\rightarrow H_2 + G, \texttt{if}\ v\ e_1'\ e_2' & [\textsc{Test}] \\ &= S_2'' & \end{array}$$

  By induction $\exists\ S_2'\ st.\ S_2'' \rightarrow^+ S_2'$ and $S_1' \sim S_2'$.

– [IsProxy] In this case

$$S_1 = H_1, \texttt{isProxy}\ (\texttt{proxy}\ a) \rightarrow \texttt{true} = S_1'$$

Since there are no possible subcases we have $S_2 = H_2 + G, \texttt{isProxy}\ (\texttt{proxy}\ a) \rightarrow \texttt{true} = S_2'$ and thus $S_1' \sim S_2'$ is shown directly.

– [IsNotProxy] Similar reasoning as in the [IsProxy] case.

– [context] In this case we have

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

where $H_1, e_1 \rightarrow H_1', e_1'$. In order to have $S_1 \sim S_2$ we must have $S_2 = H_2 + G, e_3$ where $H_1 \sim_G H_2$ and $E_1[e_1] \sim_G e_3$. So it must be that $\exists E_2, e_2\ st.\ E_2[e_2] = e_3$ and $E_1 \sim_G E_2, e_1 \sim_G e_2$.
Since we have $H_1, e_1 \sim H_2 + G, e_2$ and $H_1, e_1 \rightarrow H_1', e_1'$ we can say by induction $\exists H_2', e_2', G'\ st.\ H_2 + G, e_2 \rightarrow H_2', +G', e_2'$ with $H_1', e_1' \sim H_2' + G', e_2'$.
So by the [Context] rule we have $H_2 + G, E_2[e_2] \rightarrow H_2' + G', E_2[e_2']$.

We next show that $\sim$-related states are equi-convergent.

Lemma 2. *If* $S_1 \sim S_2$ *then* $S_1 \downarrow \ \Leftrightarrow\ S_2 \downarrow$

Proof: Follows from the previous lemma.

Finally, we show that any value v is contextually equivalent to the proxy handler for that value.

Lemma 3. $\forall v.\ v \cong (\texttt{HR}_v)$

Proof: We note that for any context C,

$$\emptyset, C[v] \ \sim\ \emptyset, C[\texttt{HR}_v]$$

The proof then follows from the previous lemma.