Constraining Delimited Control with Contracts*

Asumu Takikawa[1], T. Stephen Strickland[2], and Sam Tobin-Hochstadt[1]

[1] Northeastern University
[2] University of Maryland, College Park

Abstract. Most programming languages provide abstractions for non-local control flow and access to the stack by using continuations, coroutines, or generators. However, their unrestricted use breaks the local reasoning capability of a programmer. Gradual typing exacerbates this problem because typed and untyped code co-exist. We present a contract system capable of protecting code from control flow and stack manipulations by unknown components. We use these contracts to support a gradual type system, and we prove that the resulting system cannot blame typed components for errors.

1 Ubiquitous Continuations

Delimited continuations [6, 10, 12, 18, 19, 26, 27] enable the expression of many useful programming constructs such as coroutines, engines, and exceptions as libraries. Their expressive power stems from three key operations on the control stack: (1) marking a stack frame with a *prompt*; (2) *jumping* to a marked frame, discarding the context in between; and (3) *re-attaching* the slice of the control stack that the jump discarded. Continuations are not the only operations that manipulate the stack. In particular, continuation marks [4] provide the ability to (4) *annotate* a stack frame with data that can be dynamically accessed and updated from subsequent frames. They are used to implement features like general stack inspection for debugging, dynamic binding, and aspect-oriented programming as libraries [4, 22, 23].

Many dynamically-typed languages support delimited continuations and related control operators such as coroutines or generators [15, 20], and some also support continuation marks [5, 15]. Their lack of static typing, however, implies that a programmer could easily misuse manipulations of the stack to jump to the wrong place or annotate a frame with the wrong kind of data. Gradual typing addresses just these kinds of problems. Gradually typed languages allow programmers to type parts of their programs statically but leave other parts untyped. Even better, they provide strong dynamic guarantees about the safety of the combination of typed and untyped code [24, 32]. In particular, a gradually typed language does not allow untyped code to cause a run-time violation of the type invariants in the typed code.

Unfortunately, naïvely combining delimited continuations, continuation marks, and gradual typing fails to maintain the benefits of gradual typing. The numerous type systems proposed for delimited continuations [2, 6, 11, 18, 20, 21] can prevent an ill-typed

* Supported in part by NSF CRI-0855140, SHF-1064922, CCF-0915978, the Mozilla Foundation, and the DARPA CRASH program

re-attachment of a continuation or an ill-typed continuation jump. However, these type systems alone are not sufficient for gradual typing, because of the need for *dynamic* enforcement. Ordinarily, gradual type systems dynamically protect a typed component from its untyped context with a contract [14] that monitors the flow of values across the boundary [31]. Continuations, however, allow an untyped component to *bypass* the contract protection at the component boundary by jumping over the contract. After the jump, the untyped code could arrive in the middle of a typed component on the control stack and deliver an ill-typed value. Similarly, a continuation mark allows untyped code to update a stack annotation in typed code with an ill-typed value. In other words, continuations and continuation marks establish illicit communication channels between components. For the invariants of the typed language to hold, these channels require additional protection [9].

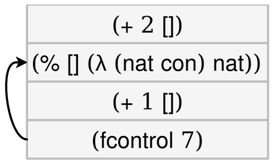
In this paper, we equip a gradually typed language with typed delimited control operators and continuation marks while maintaining the soundness of the entire system. To support this gradual type system, we introduce and formalize *control contracts* that mediate continuation jumps between prompts and their clients. We implement them in the Racket programming language [15] using *control chaperones* based on Strickland et al. [28]'s chaperone framework. Control chaperones allow a programmer to redirect communication between a prompt and a corresponding jump, inserting contract checks in between. For continuation marks, we offer an analogous pair of *continuation mark key contracts* and *continuation mark key chaperones*.

We also prove a soundness theorem for the combined language using Dimoulas et al's *complete monitoring* [9] technique. The key idea is to split a program into typed and untyped components via ownership annotations on values. Using these annotations, we impose a single owner policy which ensures that, at any given point, all of the values in the program are owned *only* by the typed or untyped portion of the program. Components may transfer ownership of a value only through the use of a contract, guaranteeing that no value changes hands without being checked. We prove that our contract system is a complete monitor and use this result to show that the gradual type system is sound.

2 Types and contracts for control operators

To illustrate how delimited continuations and continuation marks cause problems for gradual typing, we present a series of examples using Sitaram's `%` and `fcontrol` operators [26]. The following example illustrates a simple use of the `%` operator to install a prompt and then a use of `fcontrol` to jump to that prompt, aborting part of the stack. The diagram on the right depicts the control flow of the example on the stack:

```
> (+ 2 (% (+ 1 (fcontrol 7))
          (λ (nat con) (+ 1 nat))))
10
```



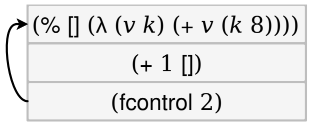
The evaluation of this example starts at `(fcontrol 7)`, which immediately discards the current continuation up to the prompt (i.e., the third frame in the diagram).

After discarding the continuation, `fcontrol` calls the handler, the λ expression argument to `%`, with two arguments: the value passed to `fcontrol` (i.e., `7`) and the discarded continuation reified as a function, i.e., `(λ (x) (+ 1 x))`. In this case, the handler just increments the first argument by one and returns, ignoring the reified continuation. The `%` operator then returns the result of the handler to its context.

The handler in this example is simple, but in general prompt handlers allow the programmer to specify arbitrary computations. The correspondence between the prompt handler and `fcontrol` matches the correspondence between exception handlers and throwing an exception [26]. In other words, continuation operators like `fcontrol` generalize exceptions [18].

One major difference between `fcontrol` and most exception interfaces is that instead of throwing the continuation away, the handler can also re-install the continuation:

```
> (% (+ 1 (fcontrol 2))
     (λ (v k) (+ v (k 8))))
11
```



Here the handler calls its second argument, the reified continuation, instead of ignoring it. Since the continuation is a value, the handler just calls it like any other function. In fact, the handler could choose to return the continuation or apply it multiple times. The presence of the reified continuation makes `fcontrol` a *higher-order* control operator, as opposed to exceptions, which usually only provide *first-order* control

2.1 Types for delimited control

To implement a type system for delimited control, we must provide a means to type-check `%` and `fcontrol`. Each handler, however, may provide a different interface to its corresponding `fcontrol`. That is, they expect different types of input from a jump. In order to give a precise type for these handlers, we need to keep different logical uses of `fcontrol` separate and type-check them separately.[3]

To distinguish prompts with conflicting uses, control operators in the literature often allow the programmer to annotate prompts with *prompt tags* [11, 16, 18, 26]. For example, an implementation of coroutines and an implementation of exceptions might both install prompts on the stack. However, the stack changes coordinated by these libraries are “logically different” [26], even if they use the same operators, and should not interfere with one another.

Prompt tags also provide a convenient means to type-check separate uses of `fcontrol` [18]. The type of a prompt tag determines the valid types of values that an application of `fcontrol` can send to the corresponding prompt’s handler. The prompt tag type also specifies the return type of the handler and the prompt’s body. The `%` and `fcontrol` operators can be used with prompt tags to allow fine-grained control over what prompt is targeted:

[3] A type and effect system for delimited control [2, 6] could provide more precise types. However, an effect system would require intrusive run-time monitoring to enforce with contracts.

```
(define handler-1 (λ (v k) (string-append v "0")))
(define handler-2 (λ (v k) (k 1)))
```

```
> (% (number->string (% (+ 1 (fcontrol "10" prompt-tag-1))
                         handler-2 prompt-tag-2))
     handler-1 prompt-tag-1)
"100"
```

Since the call to `fcontrol` uses `prompt-tag-1`, the jump arrives at the outermost prompt, which is tagged with `prompt-tag-1`. The jump triggers `handler-1`, associated with the outer prompt. Notably, the jump does *not* invoke `handler-2`. It is vital that the programmer does not use the wrong prompt tag here, because the handlers expect different types: a string for `handler-1` and an integer for `handler-2`.

Using a type system for prompt tags, we could declare that `prompt-tag-1` has the type `(Prompt String (Integer -> String) String)`. The first two types mean that the handler expects to receive a string and a function that takes integers and returns strings. The third type corresponds to the return type of the body and handler. This matches our example, since `fcontrol` sends the string `"10"` and the continuation from `fcontrol` to the outer prompt expects an integer and produces a string. Both the body (using `number->string`) and the handler clearly produce strings as well.

2.2 Gradual typing, the broken variant

In a language with gradual typing, a typed component may import unknown functions from an untyped component:

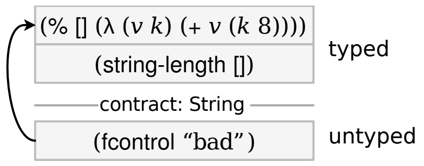
```
#lang typed/racket
(require/typed [g : Integer -> String] from "untyped.rkt")

(% (string-length (g 2))
   (λ ([v : Integer] [k : Integer -> Integer])
     (+ v (k 8))))
```

In this example, the typed component imports a function `g` that is specified to have the type `Integer -> String`, which is valid for its use in the prompt expression. The gradual type system enforces the type for `g` with the generated contract `(-> integer? string?)`. It blames the untyped component if its export fails to uphold the contract. Imports from untyped components and exports to untyped components are always protected with contracts translated from the corresponding type [31]. The type system prevents the typed component from misapplying the function.

Unfortunately, this naïve model of interaction fails in the presence of control operators, as demonstrated by the following *untyped* component:

```
#lang racket
(provide g)
(define (g x) (fcontrol "bad"))
```

The use of the `fcontrol` operator in the body of `g` immediately transfers control to the handler function when the typed module invokes `g`. Since this control transfer bypasses the contract boundary, the string `"bad"` is passed to the `+` operation, which causes a run-time failure that the type system should have prevented. The failure stems from the lack of protection on the communication channel between `fcontrol` and `%`.



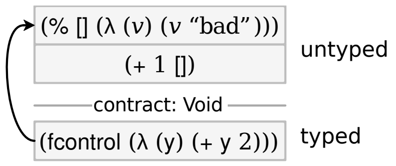
Generally speaking, the usual strategy of applying contracts to just the component imports and exports does not adequately protect the typed code from invalid uses of control operators within untyped code. In particular, the abort-like behavior of `fcon-trol` allows it to directly communicate with the handler in the typed code, without first passing through a contract check at the component boundary.

With higher-order programming, the illicit communication may also take place using control operators in the opposite direction, as in the following pair of components:

```
#lang typed/racket
(provide g)

(define: (g) : Void (fcontrol h))
(define: (h [y : Integer]) : Integer (+ 1 y))
```

```
#lang racket
(require g from "typed.rkt")
(% (+ 1 (g)) (λ (v k) (v "bad")))
```

Here, the typed component exports a function `g` that uses a control operator to jump to the prompt, passing its handler a function. The untyped component calls `g` inside of a prompt whose handler misapplies the returned function to the string `"bad"` instead of an integer. Again, we depict this situation with a diagram:



This stack illustrates a situation similar to the last diagram except that the typed and untyped components have swapped roles. Furthermore, notice that the contract on the stack is `Void` because the contract system checks the return value of `g`, because exports from typed components are wrapped with a contract.

On the surface, this may not seem like a problem; after all, the untyped component is free to do anything it likes with values since it is not beholden to a type system.

Unfortunately, `fcontrol` has smuggled the *function* `h` across the contract boundary. Since `h` originates from typed code, applying it should not cause an error that the type system could prevent. In the top frame, however, the untyped component applies `h` to a value `"bad"` that the function does not expect, causing the addition `(+ 1 y)` to fail. This shows that higher-order programming requires protection for communication in *both* directions between typed and untyped components. To make gradual typing work, we must account for and protect all extra channels of communication.

2.3 Gradual typing, fixed

In order to fix our naïve gradual type system, we reuse the key insight from the gradual typing literature: the dynamic semantics must protect *all* possible channels of communication between typed and untyped components [9]. We instantiate this research insight for stack abstractions by installing contract checks on prompt tags that activate when control operators cross component boundaries. A prompt tag is a *capability* for communicating between two stack frames in a program. Thus, only components that have access to a given tag are allowed to communicate with the matching prompt, enabling the programmer to leverage lexical scope to limit access. However, the capability nature of prompt tags only determines *who* can communicate over the channel, but not *what* can be communicated across the channel.

To enable prompt tags to protect the data communicated via control operators, we equip prompt tags with contracts that trigger when a control operator transfers a value to the matching prompt. Since prompt tags function as capabilities, a component can be assured that *only* components with access to the corresponding prompt tag can jump to its prompts. Thus, as long as typed prompt tags are always exported with appropriate contracts, other components cannot jump to them without incurring contract checks. We formally characterize the translation of types to contracts in section 4.

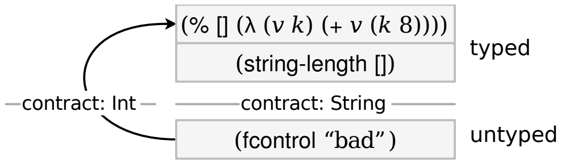
For the problematic example from before, we revise the typed component to create and export a prompt tag. The untyped component can import and use the tag to jump to the typed component's prompt:

```
#lang typed/racket
(require/typed [g : Integer -> Integer] from "untyped.rkt")
(provide pt)

(pt : (Prompt Integer (Integer -> String) Integer))
(define pt (make-prompt-tag))

(% (string-length (g 2))
   (λ ([v : Integer] [k : Integer -> Integer])
     (+ v (k 8)))
   pt)
```

As before, the prompt tag type describes the type of the two values that `fcontrol` sends to the handler and the result type of the handler. In the untyped code, the call to `fcontrol` uses the prompt tag from the typed code:

```
#lang racket
(require pt from "typed.rkt")
(provide g)
(define (g x) (fcontrol "bad" pt))
```

Now, the type system installs a contract on uses of the exported tag in untyped code that corresponds to the type `(Prompt Integer (Integer -> Integer))`. When the function `g` aborts the continuation using the `fcontrol` operator, the `"bad"` value is checked with the `Integer` contract. The contract check fails and blames the untyped component for not providing an `Integer` to the prompt's handler. Pictorially, the fix adds a second contract boundary between the use of `fcontrol` and its matching prompt:



With the second contract boundary, all possible paths between the untyped and typed components are protected. This ensures that no unmonitored communication can occur between the components. In other words, the contract system completely monitors all communication between components, thus ensuring the safety of typed code that uses continuation operations.

2.4 Continuation marks

The stack also offers non-local data storage to the programmer. Continuation marks are a language feature that enables this view, allowing the association of a key-value storage cell with each of the continuation frames that make up the stack. In turn, continuation marks enable other language features and tools such as debuggers, dynamic binding, and aspect-oriented programming [3, 4, 33].

A continuation mark is added to the current continuation frame with the `wcm` form (short for `with-continuation-mark`) and accessed with the `ccm` form (short for `current-continuation-marks`):

```
> (wcm 'key 7 (+ 1 (first (ccm 'key))))
8
```

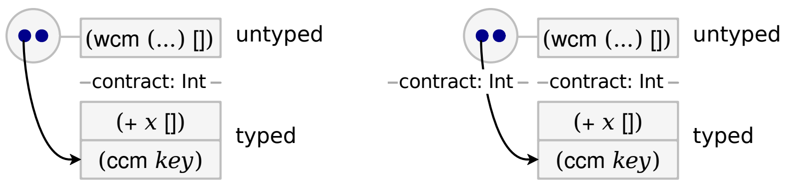
Continuation marks consist of a key and an associated value, which are passed to the `wcm` operation. The `ccm` operation returns a list of the marks stored in the continuation associated with some key. The previous example demonstrates a simple case of setting and accessing a mark. As with continuations, continuation marks allow non-local communication of data through the stack, and thus require new forms of protection from the contract system. More concretely, continuation marks can be set in an untyped component and then accessed later in a typed component:

```
#lang racket
(require g from "typed.rkt")
(define key (make-continuation-mark-key))
(wcm key "bad" (g 7))
```

```
#lang typed/racket
(require/typed [key : (Mark Integer)] from "untyped.rkt")
(provide g)
(define (g x) (+ x (first (ccm key))))
```

In this example, the untyped component stores a string in the continuation mark with a new continuation mark key. The typed component imports the key with a type that requires integers in the mark storage for the key. However, the untyped component has already violated this assumption by storing a string in the mark. This demonstrates another example of an unprotected stack-based channel of communication.

Our solution for continuation mark protection is similar to the solution for delimited continuations. First, instead of allowing any value as a key for marks, we require the use of a prompt-tag-like key type, which we call a *continuation mark key*. This key acts as a capability for accessing the data contained in the mark. Using the same technique as prompt tags, we attach contracts to this key so that the continuation mark operations can introduce contract checks based on the key's contract. The following diagram illustrates the example above (on the left) and our solution (right):



The new circle attached to the top stack frame illustrates a storage cell for the continuation mark on that frame. The cell might store many values, up to one for each key. The arrow from the cell depicts the flow of a value from the cell to the continuation frame that requests it using the `ccm` operation. As with continuations, this flow bypasses the ordinary contract boundary on the stack. The diagram on the right shows the fix in the form of an extra contract boundary that is established for accesses to the continuation mark store. In short, the contract system must protect all possible channels of interaction between the typed and untyped portions of the program.

From a contract system design perspective, continuation marks are similar to mutable reference; both enable non-local communication. Moreover, contracts for references and marks have related semantics. Mutable references need specialized support from the contract system to ensure that all access to the reference is protected by a contract [9, 28]. This extra protection amounts to wrapping the reference with a guard that redirects reads or writes to the reference and injects appropriate contract checking. Similarly, continuation mark key contracts wrap the key with a guard that redirects reads or writes to the continuation mark. Our formal model characterizes these guards and contracts more precisely.

3 Formalizing contracts for stack abstractions

To explain our design and to validate its soundness, we present a formal model of a gradually typed λ-calculus extended with low-level operations on stacks. The low-level operators faithfully macro-express [13] the high-level operators. We chose our model's operators to match the production libraries used in both Racket [16] and in Guile [17] in order to demonstrate the model's practical applicability. Further details are available in a technical report [30].

Dybvig et al. [11] identify a template of five key operations that are necessary for delimited continuations: (1) construction of a delimiter, (2) delimiting a continuation, (3) capturing a continuation, (4) aborting a continuation, and (5) re-instating a continuation. Our model provides each of the elements in the template above. In our case, (1) corresponds to prompt tag creation and (5) to function application. The remaining three are provided as distinct operations `%`, `call/comp`, and `abort`, detailed below.

Our language is Dimoulas et al's CPCF [8], extended with Flatt et al's continuation operators [16]. We augment this model with an adaptation of Gunter et al. [18]'s type system for delimited control and a type system for continuation marks based on similar ideas. For dynamic invariant enforcement, we add contracts for delimited continuations and continuation marks.

Figure 1 presents the core grammar of the model. Programs consist of a tuple with an expression and a store. The store tracks the allocation of prompt tags and continuation mark keys. Expressions include straightforward PCF operations, list operations, and a set of control operators. The language is parameterized over a set of basic data types and primitive unary and binary operations such as addition, subtraction, and so on.

The key control operators are `(%` e_1 e_2 v`)`, `(abort` e_1 e_2`)`, and `(call/comp` v e`)`, which correspond to delimiting the continuation, aborting the continuation, and capturing the continuation respectively. For continuation marks, the `(call/cm` e_1 e_2 e_3`)` and `(ccm` e`)` operations model the setting of continuation marks and access of marks respectively.

```
P ::= <e, σ>
σ ::= ∅ | (key σ) | (tag σ)
e ::= x | v | (e e) | (if e e e) | (μ (x : t) e)
    | (unop e) | (binop e e) | (cons e e)
    | (case e (null = e) ((cons x x) = e))
    | (prompt-tag) | (cm-key)
    | (% e e v) | (abort e e)
    | (wcm w e) | (ccm e)
    | (call/comp e e) | (call/cm e e e)
    | (update mk e); e
    | (error)
v ::= b | (λ (x : t) e) | pt | mk
    | (cons v v) | null
    | call/comp | call/cm

 t ::= B | (→ t t) | (Prompt t t)
     | (Mark t) | (List t)
pt ::= tag
mk ::= key
 E ::= M | (wcm w M)
 M ::= [] | (if E e e) | (E e) | (v E)
     | (unop E) | (binop E e) | (binop v E)
     | (case E (null = e) ((cons x x) = e))
     | (cons E e) | (cons v E)
     | (update mk E); e
     | (% e E v) | (% E pt v)
     | (abort E e) | (abort v E)
     | (call/comp E e) | (call/comp v E)
     | (call/cm E e e) | (call/cm v E e)
```

Figure 1: Core grammar and evaluation contexts

$$\frac{\begin{array}{c}\Gamma \mid \Sigma \vdash e_1 : t_1 \\ \Gamma \mid \Sigma \vdash e_1 : \texttt{(Prompt } t_1\ t_2\texttt{)}\end{array}}{\Gamma \mid \Sigma \vdash \texttt{(abort } e_1\ e_2\texttt{)} : t}\,[\text{TAbort}] \qquad \frac{\begin{array}{c}\Gamma \mid \Sigma \vdash e_1 : \texttt{(}\to \texttt{(}\to t_3\ t_2\texttt{)}\ t_3\texttt{)} \\ \Gamma \mid \Sigma \vdash e_2 : \texttt{(Prompt } t_1\ t_2\texttt{)}\end{array}}{\Gamma \mid \Sigma \vdash \texttt{(call/comp } e_1\ e_2\texttt{)} : t_3}\,[\text{TCallComp}]$$

$$\frac{\begin{array}{c}\Gamma \mid \Sigma \vdash e_1 : \texttt{(Mark } t_1\texttt{)} \\ \Gamma \mid \Sigma \vdash e_2 : t_1 \qquad \Gamma \mid \Sigma \vdash e_3 : t_2\end{array}}{\Gamma \mid \Sigma \vdash \texttt{(call/cm } e_1\ e_2\ e_3\texttt{)} : t_2}\,[\text{TCallCM}] \qquad \frac{\Gamma \mid \Sigma \vdash e : \texttt{(Mark } t\texttt{)}}{\Gamma \mid \Sigma \vdash \texttt{(ccm } e\texttt{)} : \texttt{(List } t\texttt{)}}\,[\text{TCCM}]$$

$$\frac{\begin{array}{c}\Gamma \mid \Sigma \vdash e_1 : t_2 \\ \Gamma \mid \Sigma \vdash v : \texttt{(}\to t_1\ t_2\texttt{)} \\ \Gamma \mid \Sigma \vdash e_2 : \texttt{(Prompt } t_1\ t_2\texttt{)}\end{array}}{\Gamma \mid \Sigma \vdash \texttt{(\% } e_1\ e_2\ v\texttt{)} : t_2}\,[\text{TPrompt}] \qquad \frac{\begin{array}{c}\Gamma \mid \Sigma \vdash e : t \\ \Gamma \mid \Sigma \vdash mk : \texttt{(Mark } t_1\texttt{)} \quad \ldots \\ \Gamma \mid \Sigma \vdash v : t_1 \quad \ldots\end{array}}{\Gamma \mid \Sigma \vdash \texttt{(wcm ((} mk\ v\texttt{)} \ldots\texttt{)}\ e\texttt{)} : t}\,[\text{TWCM}]$$

Figure 2: Typing rules

Our continuation operators suffice to encode high-level operators such as `%` and `fcontrol`. For example, here is a macro encoding of `fcontrol`:

```
(fcontrol v p) = (call/comp (λ (k) (abort v k)) p)
```

The static semantics of the model is straightforward. For delimited continuations, we adapt the prompt types of Gunter et al.'s `cupto` system [18]. The major type judgments are shown in figure 2. A judgment $\Gamma \mid \Sigma \vdash e : t$ separates the environment typing Γ from the store typing Σ. The store typing straightforwardly keeps track of the types of allocated prompt tags and mark keys. Prompt tag types (Prompt t_1 t_2) are parameterized by two types: t_1 for the argument type expected by the handler function and t_2 for the body of the prompt. The rule for prompt expressions requires that, given an appropriate prompt tag, the body and the handler both produce a result of type t_2 and that the handler accepts an argument of type t_1. Conversely, an abort must carry a value of type t_2 for a given prompt tag and may result in any type, since control never returns.

Meanwhile, the `call/comp` operator captures a continuation up to a prompt with the given prompt tag and passes it to its handler. The return type t_3 of `call/comp` is the return type of its argument function. Since the current continuation has a hole of type t_3, and since the type of the expression up to the prompt is dictated by the prompt tag type t_2, the type rule also requires that `call/comp`'s argument expects an argument type of (→ t_3 t_2).

Continuation mark keys have a type (Mark t) where t is type of the value to be stored in the mark. The rule for `wcm` requires that all of the key-value pairs it stores are consistently typed; that is, the value stored is well-typed with respect to the mark key's type parameter. Similarly, `call/cm` requires that the specified mark key and value match and that its result type is the result type of its body. The `ccm` operation, used to extract the mark values, returns a list containing the values of the type stored in the mark.

$$
\begin{array}{rcl}
e & ::= & \ldots. \\
  & \mid & (\texttt{mon}_{l}^{l,l}\ ctc\ e) \\
  & \mid & (\texttt{ctc-error}_{l}^{l}) \\
  & \mid & (\texttt{check}_{l}^{l}\, e\ v) \\
pt & ::= & \ldots. \\
  & \mid & (\texttt{PG}_{l}^{l,l}\ ctc\ pt) \\
mk & ::= & \ldots. \\
  & \mid & (\texttt{MG}_{l}^{l,l}\ ctc\ mk) \\
ctc & ::= & (\texttt{flat}\ (\lambda\ (x : t)\ e)) \\
  & \mid & (\mapsto ctc\ ctc) \\
  & \mid & (\texttt{prompt-tag/c}\ ctc) \\
  & \mid & (\texttt{mark/c}\ ctc) \\
  & \mid & (\texttt{list/c}\ ctc) \\
t & ::= & \ldots. \\
  & \mid & (\texttt{Con}\ t) \\
M & ::= & \ldots. \\
  & \mid & (\texttt{mon}_{l}^{l,l}\ ctc\ E) \\
  & \mid & (\texttt{check}_{l}^{l}\, E\ v)
\end{array}
$$

$$
\begin{array}{l}
(\texttt{mon}_{j}^{k,l}\ (\texttt{flat}\ v_f)\ v) \longrightarrow (\texttt{check}_{j}^{k}\ (v_f\ v)\ v) \\
(\texttt{mon}_{j}^{k,l}\ (\mapsto ctc_a\ ctc_r)\ v) \longrightarrow \\
(\lambda\ (x_1 : t) \\
\quad ((\lambda\ (x_2 : t)\ (\texttt{mon}_{j}^{k,l}\ ctc_r\ (v\ x_2))) \\
\quad\ (\texttt{mon}_{j}^{l,k}\ ctc_a\ x_1))) \\
\qquad\qquad \text{where } v = (\lambda\ (x : t)\ e) \\
(\texttt{mon}_{j}^{k,l}\ (\texttt{list/c}\ ctc)\ \texttt{null}) \longrightarrow \texttt{null} \\
(\texttt{mon}_{j}^{k,l}\ (\texttt{list/c}\ ctc)\ (\texttt{cons}\ v_1\ v_2)) \longrightarrow \\
(\texttt{cons}\ (\texttt{mon}_{j}^{k,l}\ ctc\ v_1)\ (\texttt{mon}_{j}^{k,l}\ (\texttt{list/c}\ ctc)\ v_2)) \\
(\texttt{mon}_{j}^{k,l}\ (\texttt{prompt-tag/c}\ ctc)\ v_p) \longrightarrow (\texttt{PG}_{j}^{k,l}\ ctc\ v_p) \\
(\texttt{mon}_{j}^{k,l}\ (\texttt{mark/c}\ ctc)\ v_m) \longrightarrow (\texttt{MG}_{j}^{k,l}\ ctc\ v_m) \\
(\texttt{check}_{j}^{l}\ \#\text{t}\ v) \longrightarrow v \\
(\texttt{check}_{j}^{l}\ \#\text{f}\ v) \longrightarrow (\texttt{ctc-error}_{j}^{l})
\end{array}
$$

Figure 3: Contracts and monitors

We specify the dynamic semantics in an operational style using evaluation contexts [12], omitting straightforward rules for conventional operations. The evaluation contexts, shown in figure 1, follow the form of the expression grammar. The contexts are stratified into two non-terminals E and M to ensure that adjacent `wcm` frames in the context are merged before further reduction. These merge steps simplify the rest of the operational rules and have precedent in the continuation mark literature [4, 16].

The contract system, based on CPCF, adds additional constructs to the language. The additional constructs and reduction rules for contracts are shown in figure 3. Contracts are applied using both monitors and guards. A monitor $(\texttt{mon}_{j}^{k,l}\ ctc\ e)$ represents a term e protected by a contract ctc. The labels k and l indicate the server and client parties, respectively, that entered into the contract. The final label j indicates the component that the contract belongs to [9]. Since monitored terms are not values, we need additional guard terms for prompt tags and continuation mark keys, because guarded tags and keys may appear in positions that expect values. Guards, like monitors, include a contract and server, client, and contract labels for the involved parties. A monitor or guard and its labels delineate the boundary between two components: server and client. Boundaries play a key role when we prove that no values pass between components (i.e., across a monitor or guard) without appropriate contract protection.

Monitors with a flat contract, i.e., one that the contract system can immediately check, reduce to a `check` expression that runs the contract predicate and either raises a contract error or returns the checked value. Monitors for functions reduce to a wrapped function that checks both the domain and range contracts. For prompt tags and mark contracts, the monitors respectively reduce to a prompt tag or mark key guard.

Figure 4 shows the key rules for continuations and continuation marks, which warrant additional explanation. The `make-prompt-tag` and `make-cm-key` terms reduce to fresh prompt tag and mark key values, respectively, allocating them in the store. A prompt that contains a value reduces to the value itself.

When a prompt contains an abort, the reduction relation takes the prompt's handler and applies it to the aborted value (via [abort]). The notation E_{pt} means that the context E does *not* contain a prompt tagged with the prompt tag pt. The rule additionally wraps the aborted value with any necessary contracts using the `wrap+` and `wrap-` metafunctions. Respectively, these metafunctions wrap the values with the contract checks that are necessitated by the prompt tag of the prompt and the tag used by the abort. This rule only triggers when the prompt tag on the prompt side and the abort side are equivalent modulo any contract guards. An abort where the surrounding context contains no matching prompt tag gets stuck. Like Gunter et al. [18], we isolate this error case, which is difficult to rule out without a type and effect system, in our theorems.

Within a prompt, `call/comp` reifies the continuation as $(\lambda\ (x : t)\ E_{pt}[x])$ and applies v to this function. As with aborts, the rule only triggers when the prompt tags on both sides are the same.

Continuation mark frames $(\texttt{wcm}\ w\ v)$ are discarded when the body is a value and reduce to the value itself. When two continuation mark frames are directly adjacent in the form $(\texttt{wcm}\ w_1\ (\texttt{wcm}\ w_2\ e))$, the frames are merged. The metafunction takes the innermost value for any given continuation mark key for the resulting store.

For continuation captures and setting a continuation mark, a continuation mark frame is allocated unless one already exists in the continuation. These are used to ensure that subsequent updates to the marks can be carried out. A `call/cm` operation that sets or updates a continuation mark reduces to an intermediate `update` term that first applies any necessary contract checks and then sets or updates the mark value. Continuation marks are actually updated via the [wcm/update/set] and [wcm/update/add] rules. The values stored in a continuation mark are extracted with the `ccm` expression for a given mark key. If the key is unguarded, the reduction rule uses a metafunction to retrieve the relevant values stored in the continuation's mark frames. If a guard exists, the `ccm` term reduces to a contract check wrapped around a new `ccm` term.

Notice that the only rules that involve both contracts and control operators are those that potentially cross into another component across a monitor or guard. Specifically, these are the [abort], [ccm/guard], and [call/cm] rules. None of the other control rules involve contracts, demonstrating the one key intuition behind our formalism: only the operations that set up communication across component boundaries need additional attention from the contract system. The proof technique in the next section justifies this intuition.

4 Complete monitoring and the Blame Theorem

To show that our contract system comprehensively protects all of the communication channels in the language, we prove that the contract system satisfies the *complete monitoring* property [9]. Essentially, this property requires that the values in the language are always owned and manipulated by a single component at a time. Values only flow to a different component under the auspices of the contract system. Expressions that attempt to smuggle values without the contract system's knowledge would get stuck. We prove that the reduction relation is a complete monitor in order to show the blame theorem, which informally states that the contract system does not find the typed component at fault for any violation of types turned into contracts.

$$\langle E[\texttt{(prompt-tag)}], \sigma\rangle \Longrightarrow \langle E[tag], \texttt{(}tag\ \sigma\texttt{)}\rangle \text{ where } tag \notin \sigma \qquad \text{[prompt-tag]}$$

$$\texttt{(\% } v_1\ pt\ v_2\texttt{)} \longrightarrow v_1 \qquad \text{[prompt]}$$

$$\texttt{(\% } E_{pt}[\texttt{(abort } pt\ v\texttt{)}]\ pt_1\ v_h\texttt{)} \longrightarrow \texttt{(}v_h\ E_+[E_-[v]]\texttt{)} \qquad \text{[abort]}$$
$$\text{where } E_+ = \text{wrap+}[\![pt_1]\!],\ E_- = \text{wrap-}[\![pt, \texttt{[]}]\!],\ pt =_{pt} pt_1$$

$$\texttt{(\% } E_{pt}[\texttt{(wcm } w \texttt{ (call/comp } v\ pt\texttt{))}]\ pt_1\ v_h\texttt{)} \longrightarrow \texttt{(\% } E_{pt}[\texttt{(wcm } w \texttt{ (}v \texttt{ (}\lambda\texttt{ (}x : t\texttt{) } E_{pt}[x]\texttt{)))}]\ pt_1\ v_h\texttt{)} \text{ where } pt =_{pt} pt_1 \qquad \text{[call/comp]}$$

$$\langle E[\texttt{(cm-key)}], \sigma\rangle \Longrightarrow \langle E[key], \texttt{(}key\ \sigma\texttt{)}\rangle \text{ where } key \notin \sigma \qquad \text{[mark-key]}$$

$$\texttt{(wcm } w\ v\texttt{)} \longrightarrow v \qquad \text{[wcm/v]}$$

$$\texttt{(wcm } w_1 \texttt{ (wcm } w_2\ e\texttt{))} \longrightarrow \texttt{(wcm (}w_1 \oplus w_2\texttt{) } e\texttt{)} \qquad \text{[wcm/merge]}$$

$$\langle E[\texttt{(call/cm } v_1\ v_2\ e\texttt{)}], \sigma\rangle \Longrightarrow \langle E[\texttt{(wcm () (call/cm } v_1\ v_2\ e\texttt{))}], \sigma\rangle \text{ where } E \neq E_1[\texttt{(wcm } w \texttt{ [])}] \qquad \text{[wcm/intro/cm]}$$

$$\texttt{(wcm } w \texttt{ (call/cm } mk\ v\ e\texttt{))} \longrightarrow \texttt{(wcm } w \texttt{ (update } mk_1\ e_1\texttt{); } e\texttt{)} \text{ where } \texttt{(}mk_1\ e_1\texttt{)} = \text{push}[\![mk, v]\!] \qquad \text{[call/cm]}$$

$$\texttt{(wcm ((}key_1\ v_1\texttt{) ... (}key_2\ v_2\texttt{) (}key_3\ v_3\texttt{) ...) (update } key_2\ v_4\texttt{); } e\texttt{)} \longrightarrow \texttt{(wcm ((}key_1\ v_1\texttt{) ... (}key_2\ v_4\texttt{) (}key_3\ v_3\texttt{) ...) } e\texttt{)} \qquad \text{[wcm/set]}$$

$$\texttt{(wcm ((}key_1\ v_1\texttt{) ...) (update } key_2\ v_2\texttt{); } e\texttt{)} \longrightarrow \texttt{(wcm ((}key_1\ v_1\texttt{) ... (}key_2\ v_2\texttt{)) } e\texttt{)} \text{ where } key_2 \notin \texttt{(}key_1 \texttt{ ...)} \qquad \text{[wcm/add]}$$

$$\langle E[\texttt{(ccm } key\texttt{)}], \sigma\rangle \Longrightarrow \langle E[\text{marks}[\![E, key, \texttt{null}]\!]], \sigma\rangle \qquad \text{[ccm]}$$

$$\texttt{(ccm (MG}_j^{k,l}\ ctc\ mk\texttt{))} \longrightarrow \texttt{(mon}_j^{k,l} \texttt{ (list/c } ctc\texttt{) (ccm } mk\texttt{))} \qquad \text{[ccm/guard]}$$

$$\text{wrap+}[\![\texttt{(PG}_j^{k,l}\ ctc\ pt\texttt{)}]\!] = \texttt{(mon}_j^{k,l}\ ctc\ \text{wrap+}[\![pt]\!]\texttt{)} \qquad \text{wrap-}[\![\texttt{(PG}_j^{k,l}\ ctc\ pt\texttt{)}, E]\!] = \text{wrap-}[\![pt, \texttt{(mon}_j^{l,k}\ ctc\ E\texttt{)}]\!]$$
$$\text{wrap+}[\![tag]\!] = \texttt{[]} \qquad \text{wrap-}[\![tag, E]\!] = E$$

Figure 4: Control reductions

Judgment	**Description**
$\Gamma;\ \Sigma;\ l \Vdash e$	Well-formed source terms
$\Gamma;\ \Sigma;\ (l\ ...);\ (l\ ...);\ l \rhd ctc$	Well-formed contracts
$\Gamma;\ \Sigma;\ l \vDash e$	Loosely well-formed terms
$\Sigma \sim \sigma$	Well-formed store
$S, S^{\upsilon} \mid G \vdash e : t$	Well-typed mixed terms (sec. 5)
$S, S^{\upsilon} \mid G \vdash e$	Well-formed mixed terms (sec. 5)

Figure 5: Judgments

$$\frac{l = \Sigma(tag)}{\Sigma;\ \Gamma;\ l \Vdash tag}\,[\text{WPromptTag}] \qquad \frac{l = \Sigma(key)}{\Sigma;\ \Gamma;\ l \Vdash key}\,[\text{WKey}] \qquad \frac{\Sigma;\ \Gamma;\ l \Vdash e}{\Sigma;\ \Gamma;\ l \Vdash |e|^{l}}\,[\text{WOwn}]$$

$$\frac{\begin{array}{c}\Sigma;\ \Gamma;\ k \Vdash e \\ \Sigma;\ \Gamma;\ (k);\ (l);\ j \rhd ctc\end{array}}{\Sigma;\ \Gamma;\ l \Vdash (\texttt{mon}_{j}^{k,l}\ ctc\ |e|^{k})}\,[\text{WMon}] \qquad \frac{\begin{array}{c}\Sigma;\ \Gamma;\ k \Vdash v \\ \Sigma;\ \Gamma;\ (k\ l);\ (k\ l);\ j \rhd ctc\end{array}}{\Sigma;\ \Gamma;\ l \Vdash (\texttt{PG}_{j}^{k,l}\ ctc\ v)}\,[\text{WPromptGuard}]$$

$$\frac{\begin{array}{c}\Sigma;\ \Gamma;\ l \Vdash e \\ \Sigma;\ \Gamma;\ \Sigma(key) \Vdash v \qquad \ldots\end{array}}{\Sigma;\ \Gamma;\ l \Vdash (\texttt{wcm}\ ((key\ v)\ \ldots)\ e)}\,[\text{WWCM}] \qquad \frac{\begin{array}{c}\Sigma;\ \Gamma;\ k \Vdash key \\ \Sigma;\ \Gamma;\ k \Vdash v \qquad \Sigma;\ \Gamma;\ l \Vdash e\end{array}}{\Sigma;\ \Gamma;\ l \Vdash (\texttt{update}\ k\ key\ v);\ e}\,[\text{WUpdate}]$$

$$\frac{\Sigma;\ \Gamma;\ j \Vdash e}{\Sigma;\ \Gamma;\ (k\ \ldots);\ (l\ \ldots);\ j \rhd \lfloor(\texttt{flat}\ |e|^{j})\rfloor^{k\ldots}}\,[\text{WCFlat}] \qquad \frac{\begin{array}{c}\Sigma;\ \Gamma;\ (l\ \ldots);\ (k\ \ldots);\ j \rhd ctc_1 \\ \Sigma;\ \Gamma;\ (k\ \ldots);\ (l\ \ldots);\ j \rhd ctc_2\end{array}}{\Sigma;\ \Gamma;\ (k\ \ldots);\ (l\ \ldots);\ j \rhd (\mapsto\ ctc_1\ ctc_2)}\,[\text{WCFun}]$$

Figure 6: Selected well-formed source program and contract rules

A proof of complete monitoring requires an annotation of values and expressions with ownership labels, using the component labels that contracts already use. In addition, we annotate contracts with obligation labels to show which components are responsible for which parts of the contract:

$$\begin{array}{lll} e ::= \ldots. & v ::= \ldots. & ctc ::= \ldots. \\ \quad \mid |e|^{l} & \quad \mid |v|^{l} & \quad \mid \lfloor(\texttt{flat}\ (\lambda\ (x : t)\ e))\rfloor^{l\ldots} \\ \quad \mid (\texttt{update}\ l\ mk\ e);\ e & & \end{array}$$

The proof that our reduction relation is a complete monitor utilizes the traditional subject reduction technique. First, we describe how to set up the subject. We use several judgments, listed in figure 5, to enforce the necessary properties from the contract system. The judgment $\Gamma;\ \Sigma;\ l \Vdash e$ checks that source programs are well-formed with respect to the ownership annotations. We omit the details of several judgments; see the separate appendix for additional rules. Figure 6 presents a key subset of the rules for our model. Essentially, the judgment ensures that terms that set up a contract boundary, i.e., monitors, guards, and so on, contain sub-terms with matching ownership. For example, a monitor must be well-formed under its server label and its sub-term must be well-formed under the monitor's client label with an appropriate annotation. Guards set up a contract boundary in a similar fashion.

The judgment also features a store environment Σ. We use this environment to statically track the ownership of prompt tags and continuation mark keys. Since these values are unique and originate in a single component, we say that their ownership is determined purely by their mapping in the store. This ensures that any given tag or mark key appears only in the component that created them *unless* transported to another component via a contract.

$$
\begin{array}{rcll}
(\|(\lambda\ (x : t)\ e)\|^{l}\ \|v\|^{l}) & \longrightarrow & |e[x := |v|^{l}]|^{l} & [\beta] \\[1em]
(\%\ E^{k}_{pt}[(\texttt{wcm}\ w\ (\texttt{call/comp}\ \|v\|^{k}\ \|pt\|^{k}))] & \longrightarrow & (\%\ E^{k}_{pt}[(\texttt{wcm}\ w\ (|v|^{k}\ |(\texttt{cont}\ E)|^{k}))] & [\text{call/comp}] \\
\quad \|pt_1\|^{l}\ \|v_h\|^{l}) & & \quad \|pt_1\|^{l}\ \|v_h\|^{l}) & \\
 & & \text{where } pt =_{pt} pt_1 & \\[1em]
(\%\ E^{k}[(\texttt{abort}\ \|pt\|^{k}\ \|v\|^{k})]\ \|pt_1\|^{l}\ \|v_h\|^{l}) & \longrightarrow & (|v_h|^{l}\ E^{j}_{+}[E^{k}_{-}[|v|^{k}]]) & [\text{abort}] \\
\multicolumn{3}{r}{\text{where } E^{j}_{+} = \text{wrap+}[\![pt_1]\!],\ E^{k}_{-} = \text{wrap-}[\![pt,\ []]\!],\ pt =_{pt} pt_1} & \\[1em]
(\texttt{wcm}\ w\ (\texttt{call/cm}\ \|mk\|^{l}\ \|v\|^{l}\ e)) & \longrightarrow & (\texttt{wcm}\ w\ (\texttt{update}\ k\ key\ e_1);\ e\ ) & [\text{call/cm}] \\
 & & \text{where } (key\ e_1\ k) = \text{push}[\![mk,\ v]\!] &
\end{array}
$$

Figure 7: Select reduction rules with annotations

In the case of monitors and guards, we also require that their contract is well-formed using the judgment $\Gamma;\ \Sigma;\ (k\ ...);\ (l\ ...);\ j \triangleright ctc$. The third and fourth parts of the judgment indicate the components that should be responsible for the positive and negative parts of a contract, respectively. The fifth label indicates the component that should own the contract. Flat contracts are well-formed when their obligations match up with the positive parties and their code matches the contract party. Function contracts swap the positive and negative obligations for the domain contract. In all other cases, we require that sub-contracts are appropriately well-formed.

For some terms in a reduction sequence, the well-formedness condition is too strict. Most commonly, terms that reduce to monitored expressions can cause well-formedness to fail, even though a few additional steps of reduction corrects this failure. To handle this situation, we extend well-formedness to a *loose* well-formedness judgment $\Gamma;\ \Sigma;\ l \vDash e$, which is preserved by reduction.

Finally, we require with the judgment $\Sigma \sim \sigma$ that the program store is well-formed with respect to the store environment, meaning all of the statically known tags and keys are allocated with the correct owners. This requirement prevents a situation where unallocated tags or keys appear in an expression or where the environment records the wrong ownership.

To guarantee that the preservation lemma actually holds, we also modify the reduction rules to propagate the ownership annotations appropriately. Figure 7 shows a subset of the revised reduction rules. We rely on the notation $\|v\|^{l}$, which means the value v may be wrapped with zero or more ownership annotations, all with the label l. In the rules, we take any possibly annotated values and replace them in the contractum with the value wrapped in a single annotation, ensuring the annotation remains in future steps. One interesting case is the [call/cm] rule. The rule utilizes a modified `push` metafunction that guides the value v through several contract boundaries to reach the component that the mark key lives in. Each boundary traversal wraps the value with an additional monitor. The modified metafunction additionally returns the final owner of the component v after wrapping, which we need to annotate the `update` term.

With the judgments in mind, we formalize the complete monitoring property.

Definition 1. *A reduction relation is a complete monitor if for all well-typed terms e_0 such that $\Sigma; \varnothing; l_0 \Vdash e_0$,*

- $<e_0, \varnothing> \rightarrow^* <v, \sigma>$, *or*
- *for all e_1 and stores σ_1 such that $<e_0, \varnothing> \rightarrow^* <e_1, \sigma_1>$, there exists an e_2 and σ_2 such that $<e_1, \sigma_1> \rightarrow <e_2, \sigma_2>$, or*
- $<e_0, \varnothing> \rightarrow^* <E_{pt}[\texttt{(abort}\ v\ pt\texttt{)}], \sigma_1>$, *or*
- $<e_0, \varnothing> \rightarrow^* <e_1, \sigma_1> \rightarrow^* <\texttt{(ctc-error}^{k}_{j}\texttt{)}, \sigma_2>$ *where e_1 is of the form* $E^l[\texttt{(mon}^{k,l}_{j}\lfloor\texttt{(flat}\ v\texttt{)}\rfloor^{l\ldots}\ |v|^k\texttt{)}]$ *and $k \in (l\ \ldots)$.*

Theorem 1. *The reduction relation $\rightarrow$ is a complete monitor.*

The proof follows a standard subject reduction strategy with two main lemmas: progress and preservation. We list the key lemmas below but omit details of the proof cases, which are similar to those presented by Dimoulas et al. [9].

Lemma 1. *For all e_0, σ_0, and Σ_0 such that $\Sigma_0; \varnothing; l \Vdash e_0$ and $\Sigma_0 \sim \sigma_0$, then either*

- $e_0 = v$,
- $e_0 = \texttt{(ctc-error}^{k}_{j}\texttt{)}$,
- *there exists an e_1 and σ_1 such that $<e_0, \sigma_0> \rightarrow <e_1, \sigma_1>$, or*
- $<e_0, \sigma_0> = <E_{pt}[\texttt{(abort}\ v\ pt\texttt{)}], \sigma_0>$.

Lemma 2. *For all $<e_0, \sigma_0> \rightarrow <e_1, \sigma_1>$ and there exists Σ_0 such that $\Sigma_0; \varnothing; l_0 \Vdash e_0$ and $\Sigma_0 \sim \sigma_0$, then for some $\Sigma_1 \supseteq \Sigma_0$, $\Sigma_1; \varnothing; l \Vdash e_1$.*

The culmination of the formalism is the Blame Theorem. Informally, the key idea of the Blame Theorem is that the contract system never blames the typed components of a mixed program for a contract error. Again, we first require some setup in order to state the theorem. The technique that we use here is detailed in Dimoulas et al. [9].

First, we set up an untyped sister language of our original typed language in order to have mixed programs. The untyped language shares the syntax and current operational semantics, but omits type annotations. Second, we isolate any stuck states that occur due to type errors and reduce them to contract errors blaming the component. For the contract system, we also require that flat contracts are picked from a pool of built-in contracts that exactly correspond to the base datatypes we use: integers, strings, etc.

In the mixed language, monitors allow the embedding of expressions from other components as before. We now limit the server and client labels to τ and υ for typed and untyped components. In other words, untyped components are embedded in a typed component with server and client labels υ and τ, respectively. For embedding in the other direction, the labels are reversed.

To ensure that components are well-formed, we require that the typed portions of any mixed program are well-typed and require that all components respect the ownership annotations as before. Furthermore, we need to guarantee that all component boundaries are protected by the correct contracts. We formalize this notion in the judgments $S, S^\upsilon \mid G \vdash e$ and $S, S^\upsilon \mid G \vdash e : t$ with store typing S, an environment S^υ for tracking untyped locations, and type environment G. Our notion of store consistency requires that untyped and typed locations are tracked disjointly [7]. These judgments rely on the mapping between types and contracts, presented in figure 8.

Finally, we can state and prove the Blame Theorem:

$$\begin{aligned}
\mathrm{T}[\texttt{(}\mapsto \mathit{ctc}_1\ \mathit{ctc}_2\texttt{)}] &= \texttt{(}\rightarrow \mathrm{T}[\mathit{ctc}_1]\ \mathrm{T}[\mathit{ctc}_2]\texttt{)} \\
\mathrm{T}[\texttt{(prompt-tag/c}\ \mathit{ctc}\texttt{)}] &= \texttt{(Prompt}\ \mathrm{T}[\mathit{ctc}]\texttt{)} \\
\mathrm{T}[\texttt{(mark/c}\ \mathit{ctc}\texttt{)}] &= \texttt{(Mark}\ \mathrm{T}[\mathit{ctc}]\texttt{)} \\
\mathrm{T}[\texttt{(list/c}\ \mathit{ctc}\texttt{)}] &= \texttt{(List}\ \mathrm{T}[\mathit{ctc}]\texttt{)}
\end{aligned}$$

Figure 8: Contract-type translation

Theorem 2. *For all untyped terms e_0 such that $\varnothing, \varnothing \mid \varnothing \vdash e_0$ and $\varnothing; \varnothing; \upsilon \Vdash e_0$, $<e_0, \varnothing>$ does not reduce to a configuration of the form $<\texttt{(ctc-error}^{\tau}_{j}\texttt{)}, \sigma>$.*

The proof follows by subject reduction, again with two main lemmas [7, 9].

5 Implementing stack protection

In addition to demonstrating the theoretical soundness of our design, we also describe its implementation in a production language.[4] Our implementation technique builds on Strickland et al's chaperone framework [28]. Chaperones act as proxies for values that behave the same as the originals, modulo additional exceptions. This allows the enforcement of a desirable property of contracts: a contracted value should behave the same as the uncontracted value except for the possibility of contract errors.

To implement our control contracts, we modified the Racket runtime system to provide additional primitive operations such as `chaperone-continuation-prompt-tag` and `chaperone-continuation-mark-key`. Both prompt tag and mark key chaperones take two function arguments that are called when continuation and continuation mark operations are used, respectively. For the prompt tag case, one function is interposed on the application of a prompt handler and the other is interposed on a continuation abort. For continuation marks, one function is interposed on retrieval from a mark and the other is interposed on insertion into a mark.

Prompt and mark operations in the runtime coordinate with chaperones by checking if the prompt tag or mark key, respectively, is a chaperone and then using the appropriate interposition function if so. The interposition function receives the aborted value in the continuation case and the stored mark value in the case of continuation marks. The result of the interposition function is then used in place of the original value. If the prompt tag or mark key is not chaperoned, the operation proceeds normally.

6 Related work

Types for delimited control We use a variation of Gunter et al. [18]'s type system for the `cupto` delimited control operator. Although their type system does not support continuation marks, it inspired our solution. The main difference is our choice of primitives:

[4] Contracts for control are available in Racket 5.3 and higher. A development version of Typed Racket supports delimited control and continuation marks.

`abort` and `call/comp` are lower-level than `cupto` [15]. In addition, our type constructors for prompt tags take two arguments instead of one. This allows the handler to return a different type than its argument.

Many type systems for delimited control, following Danvy and Filinksi, use a type and effect system [2, 6]. These type systems support result type modification and statically eliminating continuation jumps to missing prompts. Our design choices make different compromises, based on two pragmatic considerations: simplicity of the system and the difficulty of dynamically enforcing effect typing with contracts.

Gradual typing and the Blame Theorem Many researchers have constructed models of gradual typing: both functional and object-oriented [1, 24, 25, 29, 31]. The soundness theorem for gradual typing originates from Tobin-Hochstadt and Felleisen [31] and was christened the "Blame Theorem" in Wadler and Findler [34]. Our proof technique for this central theorem of gradual typing comes from Dimoulas et al. [9]. In general, the idea of complete monitoring also provides the intuition for the design of a contract system for gradual typing.

7 Conclusion

Virtually every modern programming language provides facilities for accessing and manipulating the stack, with exceptions, generators, and stack inspection as just a few examples. However, these facilities add non-local flows to programs, defeating the invariants programmers expect of their code. This problem is particularly acute in gradually typed languages, where type invariants are enforced with software contracts.

In this paper, we show that contracts, originally designed to mediate between caller and receiver, extend naturally to these non-local constructs. We equip Racket's delimited control and continuation mark operations with a gradual type system enforced at the boundaries by contracts. This system maintains type soundness in arbitrary composition with untyped code, as proved via the blame theorem. The implementation of control contracts in Racket leverages the existing chaperone framework for implementing contracts.

Acknowledgments The authors wish to thank Aaron Turon and Stephen Chang for comments on early drafts. Christos Dimoulas gave valuable advice on the formalism. Matthias Felleisen also provided his feedback and insight on continuations.

Bibliography

[1] Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for All. In *Proc. ACM Sym. Principles of Programming Languages*, pp. 201–214, 2011.

[2] Kenichi Asai and Yukiyoshi Kameyama. Polymorphic Delimited Continuations. In *Proc. Asian Sym. Programming Languages and Systems*, pp. 239–254, 2007.

[3] John Clements. *Portable and High-level Access to the Stack with Continuation Marks*. PhD dissertation, Northeastern University, 2006.

[4] John Clements, Matthew Flatt, and Matthias Felleisen. Modeling an Algebraic Stepper. In *Proc. European Sym. on Programming*, pp. 320–334, 2001.

[5] John Clements, Ayswarya Sundaram, and David Herman. Implementing Continuation Marks in Javascript. In *Proc. Wksp. Scheme and Functional Programming*, 2008.

[6] Olivier Danvy and Andrzej Filinski. Abstracting Control. In *Proc. LISP and Functional Programming*, pp. 151–160, 1990.

[7] Christos Dimoulas. *Foundations for Behavioral Higher-Order Contracts*. PhD dissertation, Northeastern University, 2012.

[8] Christos Dimoulas and Matthias Felleisen. On Contract Satisfaction in a Higher-Order World. *Trans. Programming Languages and Systems* 33(5), pp. 16:1–16:29, 2011.

[9] Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Complete Monitors for Behavioral Contracts. In *Proc. European Sym. on Programming*, pp. 214–233, 2012.

[10] Richard P. Draves. *Control Transfer in Operating System Kernels*. PhD dissertation, Carnegie Mellon University, 1994.

[11] Kent Dybvig, Simon Peyton-Jones, and Amr Sabry. A Monadic Framework for Delimited Continuations. *J. Functional Programming* 17(6), pp. 687–730, 2007.

[12] Matthias Felleisen. The Theory and Practice of First-Class Prompts. In *Proc. ACM Sym. Principles of Programming Languages*, pp. 180–190, 1988.

[13] Matthias Felleisen. On the Expressive Power of Programming Languages. *Science of Computer Programming* 17(1–3), pp. 35–75, 1991.

[14] Robert Bruce Findler and Matthias Felleisen. Contracts for Higher-Order Functions. In *Proc. ACM Intl. Conf. Functional Programming*, pp. 48–59, 2002.

[15] Matthew Flatt and PLT. Reference: Racket. PLT Inc., PLT-TR-2010-1, 2010. `http://racket-lang.org/tr1/`

[16] Matthew Flatt, Gang Yu, Robert Bruce Findler, and Matthias Felleisen. Adding Delimited and Composable Control to a Production Programming Environment. In *Proc. ACM Intl. Conf. Functional Programming*, pp. 165–176, 2007.

[17] Free Software Foundation. Guile Reference Manual: Prompts. 2012. `http://www.gnu.org/software/guile/manual/html_node/Prompts.html`

[18] Carl A. Gunter, Remy Didier, and Jon G. Riecke. A Generalization of Exceptions and Control in ML-like Languages. In *Proc. ACM Intl. Conf. Functional Programming Languages and Computer Architecture*, pp. 12–23, 1995.

[19] Robert Hieb, R. Kent Dybvig, and Claude W. Anderson. Subcontinuations. *LISP and Symbolic Computation*, pp. 83–110, 1994.

[20] Roshan P. James and Amr Sabry. Yield: Mainstream Delimited Continuations. In *Proc. Theory and Practice of Delimited Continuations*, pp. 20–32, 2011.

[21] Oleg Kiselyov and Chung-chieh Shan. A Substructural Type System for Delimited Continuations. In *Proc. Intl. Conf. Typed Lambda Calculi and Applications*, pp. 223–239, 2007.

[22] Oleg Kiselyov, Chung-chieh Shan, and Amr Sabry. Delimited Dynamic Binding. In *Proc. ACM Intl. Conf. Functional Programming*, pp. 26–37, 2006.

[23] Greg Pettyjohn, John Clements, Joe Marshall, Shriram Krishnamurthi, and Matthias Felleisen. Continuations from Generalized Stack Inspection. In *Proc. ACM Intl. Conf. Functional Programming*, pp. 216–227, 2005.

[24] Jeremy G. Siek and Walid Taha. Gradual Typing for Functional Languages. In *Proc. Wksp. Scheme and Functional Programming*, 2006.

[25] Jeremy G. Siek and Walid Taha. Gradual Typing for Objects. In *Proc. European Conf. Object-Oriented Programming*, pp. 2–27, 2007.

[26] Dorai Sitaram. Handling Control. In *Proc. ACM Conf. Programming Language Design and Implementation*, pp. 147–155, 1993.

[27] Dorai Sitaram and Matthias Felleisen. Control Delimiters and their Hierarchies. *LISP and Symbolic Computation*, pp. 67–99, 1990.

[28] T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. Chaperones and Impersonators: Run-time Support for Reasonable Interposition. In *Proc. ACM Conf. Object-Oriented Programming, Systems, Languages and Applications*, 2012.

[29] Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Gradual Typing for First-Class Classes. In *Proc. ACM Conf. Object-Oriented Programming, Systems, Languages and Applications*, 2012.

[30] Asumu Takikawa, T. Stephen Strickland, and Sam Tobin-Hochstadt. Constraining Delimited Control with Contracts. Northeastern University, NU-CCIS-13-01, 2013.

[31] Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage Migration: from Scripts to Programs. In *Proc. Dynamic Languages Symposium*, pp. 964–974, 2006.

[32] Sam Tobin-Hochstadt and Matthias Felleisen. The Design and Implementation of Typed Scheme. In *Proc. ACM Sym. Principles of Programming Languages*, pp. 395–406, 2008.

[33] David B. Tucker and Shriram Krishnamurthi. Pointcuts and Advice in Higher-Order Languages. In *Proc. Intl. Conf. on Aspect-Oriented Software Development*, pp. 158–167, 2003.

[34] Philip Wadler and Robert Bruce Findler. Well-typed Programs Can't be Blamed. In *Proc. European Sym. on Programming*, pp. 1–15, 2009.