The Blame Theorem for a Linear Lambda Calculus with Type Dynamic

Technical Report

Luminous Fennell and Peter Thiemann

University of Freiburg, Georges-Köhler-Allee 079, 79110 Freiburg, Germany,
{fennell,thiemann}@informatik.uni-freiburg.de

Abstract. Scripting languages have renewed the interest in languages with dynamic types. For various reasons, realistic programs comprise dynamically typed components as well as statically typed ones. Safe and seamless interaction between these components is achieved by equipping the statically typed language with a type `Dynamic` and coercions that map between ordinary types and `Dynamic`. In such a gradual type system, coercions that map from `Dynamic` are checked at run time, throwing a blame exception on failure.
This paper enlightens a new facet of this interaction by considering a gradual type system for a linear lambda calculus with recursion and a simple kind of subtyping. Our main result is that linearity is orthogonal to gradual typing. The blame theorem, stating that the type coercions always blame the dynamically typed components, holds in a version analogous to the one proposed by Wadler and Findler, also the operational semantics of the calculus is given in a quite different way. The significance of our result comes from the observation that similar results for other calculi, e.g., affine lambda calculus, standard call-by-value and call-by-name lambda calculus, are straightforward to obtain from our results, either by simple modification of the proof for the affine case, or, for the latter two, by encoding them in the linear calculus.

Keywords: linear typing, gradual typing, subtyping

This is the technical report. It extends the submitted paper with the omitted details mentioned in the original text in an appendix. In particular, all proofs are included.

Table of Contents

1 Introduction

Many of today's computing systems contain multi-core processors or regularly access distributed services. The behavior of such a concurrent system depends on the interaction of its components, so it is important for its construction and maintenance to specify and enforce communication protocols and security policies.

Ideally, static analysis should guarantee the adherence of a system to protocols and security policies, but such a comprehensive analysis is not always possible. For example, there may be legacy components that were implemented before a specification was established, or there may be components implemented in a dynamic language that are impractical to analyze. These are situations, where it is desirable to have the ability to combine static and dynamic enforcement of specifications.

Gradual typing is a specification method that satisfies these requirements. Gradual type systems were developed to introduce typing into dynamic languages to improve the efficiency of their execution as well as their maintainability. They are also useful to mediate accesses between statically typed and dynamically typed program components [5,10,12,22,24]. A commonality of these approaches is that they extend the statically typed language with a type `Dynamic`

along with type coercions that map into and out of the `Dynamic` type. The dynamically typed part is then modeled as a program where every result is coerced to `Dynamic` and where every elimination form is preceded by a coercion from `Dynamic`. Clearly, the mapping into the `Dynamic` type never fails whereas the mapping from the `Dynamic` type to a, say, function type may fail if the actual dynamic value happens to be an integer. The standard implementation of a dynamic value is a pair consisting of a value and an encoding of its run-time type. The coercion from `Dynamic` to another type checks the run-time type against the expected one and throws an exception if the types do not match.

As a first step towards gradual enforcement of communication policies as embodied by work on session types [18], this paper considers the interaction of the type `Dynamic` and blame assignment with linearity [9]. A linear calculus formalizes single-use resources in the sense that each introduced value can and must only be eliminated exactly once. Thus, linearity is an important property in all type systems that are related to managing stateful resources, in particular communication channels in a session calculus [7, 20].

A linear lambda calculus has two dimensions that the type `Dynamic` may address: it may hide and dynamically check the type structure or the linearity requirement (or both). In this paper, we let the type system enforce linearity statically and have the type `Dynamic` mostly hide the type structure. The resulting calculus extends Turner and Wadler's linear lambda calculus [21] with cast expressions inspired by Wadler and Findler's blame calculus [22], as demonstrated in the following example program. It first wraps a pair of a linear function and a replicable function (the exponential in $!\lambda b.\ b - 1$ introduces a replicable value) into a dynamic value. Then it unwraps the components and demonstrates their use as functions and replicable functions.

$$\begin{array}{l}
\texttt{let } p = \langle D \Leftarrow (\mathit{int} \multimap \mathit{int}) \otimes\, !(\mathit{int} \multimap \mathit{int})\rangle((\lambda a.\ a + 2) \otimes (!\lambda b.\ b - 1)) \texttt{ in} \\
\texttt{let } f \otimes g_1 = \langle D \otimes D \Leftarrow D\rangle p \texttt{ in} \\
\texttt{let } !g = \langle !(\mathit{int} \multimap \mathit{int}) \Leftarrow D\rangle g_1 \texttt{ in} \\
\quad g\ (g\ (\langle \mathit{int} \multimap \mathit{int} \Leftarrow D\rangle f\ 0))
\end{array}$$

The next example illustrates a possible interaction of typed and untyped code through a latently typed communication channel, which is modeled with a linear, dynamic type. Initially, the channel is represented by a linear function $\texttt{submit} : \mathit{int} \multimap D$: it expects an integer and its remaining behavior is `Dynamic`. To use it after sending the integer 5, we attempt to coerce the remaining behavior to a $\mathit{string} \multimap \star$ channel. An exception will be raised if it turns out that the run-time type of `submit'` does not have this type.

$$\texttt{let submit'} = \langle \mathit{string} \multimap \star \Leftarrow D\rangle(\texttt{submit } 5) \texttt{ in submit'}\ \textit{"Hello"}$$

Our calculus has a facility to hide part of a linearity requirement. To this end, we extend the compatibility relation to admit casts that shortcut the elimination of the exponential, effectively allowing dynamic replicable values to be treated as linear. With this extension, the above example also executes correctly if the

$$
\begin{array}{rrl}
\text{linear variables } a, b ::= & \ldots & \text{non-linear variables } x, y ::= \quad \ldots \\
\text{types} \qquad t ::= & \star \mid t \otimes t \mid t \multimap t \mid\ !t & \\
\text{expressions} \quad e, f ::= & \star \mid \texttt{let}\ \star = e\ \ \texttt{in}\ e \mid e \otimes e \mid \texttt{let}\ a \otimes a = e\ \ \texttt{in}\ e \mid & \\
& \lambda a.\ e \mid e\ e \mid a \mid\ !(x = e) \mid \texttt{let}\ !x = e\ \ \texttt{in}\ e \mid x & \\
\text{storable values} \quad s ::= & \star \mid a \otimes a \mid \lambda a.\ e \mid\ !x &
\end{array}
$$

Fig. 1. Original type and expression syntax.

run-time type of `submit'` has a form like $!(!\mathit{string} \multimap \star)$. In both cases, we show that linearity and gradual typing are orthogonal. We first define a suitable linear lambda calculus with a gradual type system and blame assignment in the manner of Taha and Siek [14, 16] and Wadler and Findler [22], respectively. We prove the soundness of this system and then follow Wadler and Findler in formulating subtyping relations and proving a blame theorem.

Overview

Section 2 defines the static and dynamic semantics of a typed linear lambda calculus with recursion.

Section 3 contains our **first contribution**, an extension of the calculus with gradual types, which adds a type `Dynamic` and cast expressions to map between ordinary types and `Dynamic`. We prove type soundness of this extended calculus. Our gradual extension is more modular than Wadler and Findler's blame calculus [22] thus making our calculus easier to extend and making our approach easier to apply to other calculi. Section 4 recalls the subtyping relation of the blame calculus and adapts it to the linearly typed setting.

Section 5 contains our **second contribution**. We state and prove the blame theorem for the linear blame calculus.

Section 6 contains our **third contribution**. We show how to exploit the modularity by integrating shortcut casts in the system that enable direct casting from a replicable value wrapped in the dynamic type to the underlying type. Type soundness and the blame theorem still hold for this extension.

All proofs may be found in a companion technical report.[1]

2 Linear Lambda Calculus

Turner and Wadler's linear lambda calculus [21] is the basis of our investigation. Figure 1 contains the syntax of their original calculus extended to deal with recursive exponentials. Types are unit types, pairs, linear functions, and exponentials. The term syntax is based on two kinds of variables, linear ones a and non-linear ones x. There are introduction and elimination forms for unit

[1] http://proglang.informatik.uni-freiburg.de/projects/gradual/

$$
\begin{array}{rl}
\text{heaps} & H ::= \ \cdot \mid H, x = e \mid H, a = s \\
\text{evaluation contexts} & E ::= [\,] \otimes e \mid a \otimes [\,] \mid [\,]\ e \mid a\ [\,] \mid \\
& \quad \texttt{let}\ a \otimes b = [\,]\ \ \texttt{in}\ e \mid \texttt{let}\ !x = [\,]\ \ \texttt{in}\ e \mid \texttt{let}\ \star = [\,]\ \ \texttt{in}\ e
\end{array}
$$

$\boxed{\{H\}e \to \{H\}e'}$

$$
\begin{array}{rlll}
\star\text{-}I & \{H\}\star & \to & \{H, a^* = \star\}a^* \\
\star\text{-}E & \{H, a = \star\}\texttt{let}\ \star = a\ \ \texttt{in}\ e & \to & \{H\}e \\
\otimes\text{-}I & \{H\}a_1 \otimes a_2 & \to & \{H, b^* = a_1 \otimes a_2\}b^* \\
\otimes\text{-}E & \{H, a'' = a' \otimes b'\}\texttt{let}\ a \otimes b = a''\ \ \texttt{in}\ e & \to & \{H\}e[a/a', b/b'] \\
\multimap\text{-}I & \{H\}\lambda a.\ e & \to & \{H, b^* = \lambda a.\ e\}b^* \\
\multimap\text{-}E & \{H, b = \lambda a.\ e\}b\ a' & \to & \{H\}e[a/a'] \\
!\text{-}I & \{H\}!(x = e) & \to & \{H, b^* = !y^*, y^* = e[x/y^*]\}b^* \\
!\text{-}E & \{H, b = !y\}\texttt{let}\ !x = b\ \ \texttt{in}\ e & \to & \{H\}e[x/y] \\
Var & \{H, x = e\}x & \to & \{H, x = e\}e
\end{array}
$$

$$
\frac{\{H\}e \to \{H'\}e'}{\{H\}E[e] \to \{H'\}E[e']}\ \mathit{Context}
$$

Fig. 2. Original operational semantics.

values $\star$, linear pairs $e \otimes e$, and lambda abstractions. By default all expressions are linear. Following the work on Lily [6] and in this respect generalizing Turner and Wadler, replicable and recursive expressions can be constructed with the exponential $!(x = e)$, where the non-linear variable x may appear in e. The exponential can be seen as a suspended computation which, when forced, may return some value. We may also write $!e$ for $!(x = e)$ when x does not occur in e. The elimination forms, except function application, are expressed using suitable `let`-forms. Our presentation is based on Turner and Wadler's second, heap-based semantics. For that reason, our evaluation rules always return linear variables (heap pointers) that refer to storable values s. The latter only consist of introduction forms applied to suitable variables. In particular, the storable value for the exponential refers to a non-linear variable that keeps the real payload that is repeatedly evaluated.

The corresponding original operational semantics is given in Figure 2. The reduction rules are defined in terms of heap-expression configurations $\{H\}e$. The heap contains two kinds of bindings. Linear variables are bound to linear values, whereas standard variables are bound to expressions. The intention is that a linear variable is used exactly once and removed after use, whereas a standard variable can be used many times and its binding remains in the heap. In the rules we use the notation a^* to denote that a should be a fresh variable.

Linear type environments Γ,Δ and non linear type environments Θ

$$\Gamma, \Delta ::= \ \text{-} \ \mid \Gamma, a : t \qquad \Theta ::= \ \text{-} \ \mid \Theta, x : t$$

$\boxed{\Theta; \Gamma \vdash e : t}$

$$\frac{}{\Theta, x : t;\ \text{-} \vdash x : t}\ \mathit{Var} \qquad \frac{}{\Theta; a : t \vdash a : t}\ \mathit{LVar}$$

$$\frac{}{\Theta;\ \text{-} \vdash \star : \star}\ \star{-}I \qquad \frac{\Theta; \Gamma \vdash e : \star \quad \Theta; \Delta \vdash f : t}{\Theta; \Gamma, \Delta \vdash \mathtt{let}\ \star = e\ \ \mathtt{in}\ f : t}\ \star{-}E$$

$$\frac{\Theta; \Gamma \vdash e : t_e \quad \Theta; \Delta \vdash f : t_f}{\Theta; \Gamma, \Delta \vdash e \otimes f : t_e \otimes t_f}\ \otimes{-}I \qquad \frac{\Theta; \Gamma \vdash e : t_a \otimes t_b \quad \Theta; \Delta, a : t_a, b : t_b \vdash f : t_f}{\Theta; \Gamma, \Delta \vdash \mathtt{let}\ a \otimes b = e\ \ \mathtt{in}\ f : t_f}\ \otimes{-}E$$

$$\frac{\Theta; \Gamma, a : t_a \vdash e : t_e}{\Theta; \Gamma \vdash \lambda a.\ e : t_a \multimap t_e}\ \multimap{-}I \qquad \frac{\Theta; \Gamma \vdash e : t_f \multimap t_e \quad \Theta; \Delta \vdash f : t_f}{\Theta; \Gamma, \Delta \vdash e\ f : t_e}\ \multimap{-}E$$

$$\frac{\Theta, x : t;\ \text{-} \vdash e : t}{\Theta;\ \text{-} \vdash !(x = e) : !t}\ !{-}I \qquad \frac{\Theta; \Gamma \vdash e : !t_e \quad \Theta, x : t_e; \Delta \vdash f : t_f}{\Theta; \Gamma, \Delta \vdash \mathtt{let}\ !x = e\ \ \mathtt{in}\ f : t_f}\ !{-}E$$

Fig. 3. Original linear type system.

$\boxed{\vdash H : \Theta; \Gamma}$

$$\frac{}{\vdash \cdot :\ \text{-}\ ;\ \text{-}}\ \mathit{Empty}$$

$$\frac{\vdash H : \Theta; \Gamma \quad \Theta, x : t;\ \text{-} \vdash e : t}{\vdash H, x = e : \Theta, x : t; \Gamma}\ \mathit{Closure} \qquad \frac{\vdash H : \Theta; \Gamma, \Delta \quad \Theta; \Delta \vdash s : t}{\vdash H, a = s : \Theta; \Gamma, a : t}\ \mathit{Value}$$

Fig. 4. Heap typing.

Figure 3 recalls the inference rules of the corresponding type system as presented by Turner and Wadler [21]. There are two kinds of type environments, a linear one, Γ, and a non-linear one, Θ. Linearity is enforced by splitting the linear type environment in the inference rules in the usual way and by requiring the linear environment to be empty in the usual places (constants, non-linear variables, and the exponential). In addition to the expression typing, there is a heap typing judgment of the form $\vdash H : \Theta; \Gamma$ that relates a heap H to environments Θ and Γ that provide bindings for non-linear and linear variables, respectively. The judgment is defined in Figure 4 and is a minor modification of the corresponding judgment of Turner and Wadler with the *Closure* rule adapted to accomodate recursive bindings. It is needed for the type soundness proof.

The type of a heap is a pair of a linear and a non-linear type environment reflecting the respective variable bindings in the heap. Linear variables are defined by linear values, non-linear variables are defined by expressions that have no free

$$
\begin{array}{rrcl@{\qquad}rrcl}
\text{types} & t & ::= & \ldots \mid D & \text{expressions} & e & ::= & \ldots \mid \langle t \Leftarrow t \rangle^p\ e \\
\text{blame labels} & p, q & ::= & \ldots & \text{storable values} & s & ::= & \ldots \mid D_c(a) \\
\text{constructors} & c & ::= & \star \mid \otimes \mid \multimap \mid\ ! & & & & \\
\end{array}
$$

Fig. 5. Syntactic extensions.

$\boxed{\Theta; \Gamma \vdash e : t}$, $\boxed{\Theta; \Gamma \vdash s : t}$

$$\frac{\Theta; \Gamma \vdash e : t_2 \quad t_1 \sim t_2}{\Theta; \Gamma \vdash \langle t_1 \Leftarrow t_2 \rangle^p e : t_1}\ \mathit{Cast}$$

$$\frac{}{\Theta; a : \star \vdash D_\star(a) : D}\ \mathit{DynVal}\text{ - }\star \qquad \frac{}{\Theta; a : D \otimes D \vdash D_\otimes(a) : D}\ \mathit{DynVal}\text{ - }\otimes$$

$$\frac{}{\Theta; a : D \multimap D \vdash D_\multimap(a) : D}\ \mathit{DynVal}\text{ - }\multimap \qquad \frac{}{\Theta; a : !D \vdash D_!(a) : D}\ \mathit{DynVal}\text{ - }!$$

Compatibility $\boxed{t \sim t'}$

$$\frac{}{\star \sim \star} \qquad \frac{}{t \sim D} \qquad \frac{}{D \sim t} \qquad \frac{t_1 \sim t_1' \quad t_2 \sim t_2'}{t_1 \otimes t_2 \sim t_1' \otimes t_2'} \qquad \frac{t_1 \sim t_1' \quad t_2 \sim t_2'}{t_1 \multimap t_2 \sim t_1' \multimap t_2'} \qquad \frac{t \sim t'}{!t \sim !t'}$$

Fig. 6. Type system extensions and compatibility.

linear variables. Note that the type system for expressions is syntax-directed and that the heap typing rules are invertible as in Turner and Wadler's original work.

3 Linear Lambda Calculus with Type Dynamic

To introduce dynamic and gradual typing into the calculus, we extend the Turner-Wadler calculus with new constructs adapted from Wadler and Findler's work [22] as shown in Figure 5. There is a new kind of wrapper value $D_c(a)$, which denotes a dynamic value pointing to a variable a with type constructor c, a new type D of dynamic values, and a cast expression $\langle t' \Leftarrow t \rangle^p\ e$. The cast is supposed to transform the type of e from t to t'. Casts are annotated with a blame label p to later identify the place that caused a run-time error resulting from the cast. A plain blame label p indicates *positive blame* whereas an inverted label $\bar{p}$ indicates *negative blame*. Inversion is involutory so that $\bar{\bar{p}} = p$. The wrapper values are needed by the casts to perform run-time type checking. Unlike the wrapper values of Wadler and Findler's blame calculus, they are only defined for linear variables and occur exclusively as heap values.

The extensions to the type system are also inspired by the blame calculus. The additional rules are given in Figure 6. The *Cast* rule changes the type t_2 of an underlying expression to a compatible type t_1. The compatibility relation $t \sim t$ is also defined in Figure 6. It is analogous to the definition in the blame

calculus. However, as subset types are not considered here, the only possible casts are either trivial or cast from or to D. The four value rules are only used inside of heap typings.

Figure 7 shows the reduction rules that perform run-time type checking by manipulating casts and wrapper values. These rules are original to our system and they implement casting in a different way than Wadler and Findler. The first set of rules concerns casting into the dynamic type. It distinguishes casts by checking the source type for groundness (a type constructor directly applied to D). A cast from a ground type to D corresponds to a wrapper application (second part of first group) whereas any other cast is decomposed into a wrapper cast *preceded* by a cast into the ground type. It is not possible to merge these two parts because a wrapper cannot be applied before its argument is properly casted and evaluated.

The second group of rules specifies the functorial casts that descent structurally in the type. For the types dynamic and unit, the casts do nothing. For pairs, the casts are distributed to the components. For functions, the cast is turned into a wrapper function, which casts the argument with types reversed and the blame label "inverted" and which casts the result with the types in the original order. A cast between exponential types unfolds the exponential and performs the cast on the underlying linear value.

The third group of rules concerns casting from type D. This group has a simpler structure than the first group because top-level unwrapping can be done in any case. The "remaining" cast is then applied to the unwrapped value of ground type. It is put to work by the functorial rules.

The last group defines the failure rules. They apply to unwrapping casts that are applied to dynamic values with the wrong top type constructor. These are the only rules that generate blame.

The resulting linear blame calculus is still type safe. We can prove preservation and progress lemmas extending the ones given by Turner and Wadler.

Lemma 1 (Type preservation). *If* $\{H\}e \to \{H'\}e'$ *and* $\vdash H : \Theta; \Gamma$ *and* $\Theta; \Gamma \vdash e : t$ *then there exist* Θ' *and* Γ' *such that* $\vdash H' : \Theta, \Theta'; \Gamma'$ *and* $\Theta, \Theta'; \Gamma' \vdash e' : t$.

Lemma 2 (Progress). *If* $\vdash H : \Theta; \Gamma$ *and* $\Theta; \Gamma \vdash e : t$ *then one of the following alternatives holds: (i) there exist* H' *and* e' *such that* $\{H\}e \to \{H'\}e'$, *(ii) there exists* p *such that* $\{H\}e \to \Uparrow p$, *or (iii)* e *is a linear variable.*

The last case may be surprising, as the standard formulation of a progress lemma states the outcome of a value at this point. However, in the present setting, all values are represented by (linear) pointers to the heap. Thus, the presence of a linear variable indicates that the evaluation returns the linear value pointed to by the variable.

From these two lemmas, we can prove type soundness in the usual way [23].

Theorem 1 (Type soundness). *If* $\vdash H : \Theta; \Gamma$ *and* $\Theta; \Gamma \vdash e : t$ *then either (i)* $\{H\}e$ *diverges, (ii)* $\{H\}e \to^* \Uparrow p$, *for some* p, *or (iii)* $\{H\}e \to^* \{H'\}a$, *for some* H' *and* a.

$$\begin{array}{rrl} \text{results} & r ::= & \{H\}e \mid \ \Uparrow p \\ \text{eval. contexts} & E ::= & \ldots \mid \langle t_1 \Leftarrow t_2 \rangle^p [\,] \end{array}$$

$\boxed{\{H\}e \to r}$

$$\frac{\{H\}e \to \Uparrow p}{\{H\}E[e] \to \Uparrow p}\ \mathit{ContextFail}$$

Casting to D

$$\begin{array}{rlll} \mathit{CastDyn}\text{ - }\otimes & \{H\}\langle D \Leftarrow t_1 \otimes t_2\rangle^p a & \to & \{H\}\langle D \Leftarrow D \otimes D\rangle^p \langle D \otimes D \Leftarrow t_1 \otimes t_2\rangle^p a \\ & & & \text{if } t_1 \neq D \vee t_2 \neq D \\ \mathit{CastDyn}\text{ - }\multimap & \{H\}\langle D \Leftarrow t_1 \multimap t_2\rangle^p a & \to & \{H\}\langle D \Leftarrow D \multimap D\rangle^p \langle D \multimap D \Leftarrow t_1 \multimap t_2\rangle^p a \\ & & & \text{if } t_1 \neq D \vee t_2 \neq D \\ \mathit{CastDyn}\text{ - }! & \{H\}\langle D \Leftarrow !t\rangle^p a & \to & \{H\}\langle D \Leftarrow !D\rangle^p \langle !D \Leftarrow !t\rangle^p a \\ & & & \text{if } t \neq D \end{array}$$

$$\begin{array}{rlll} \mathit{WrapDyn}\text{ - }\star & \{H\}\langle D \Leftarrow \star\rangle^p a & \to & \{H, b^* = D_\star(a)\}b^* \\ \mathit{WrapDyn}\text{ - }\otimes & \{H\}\langle D \Leftarrow D \otimes D\rangle^p a & \to & \{H, b^* = D_\otimes(a)\}b^* \\ \mathit{WrapDyn}\text{ - }\multimap & \{H\}\langle D \Leftarrow D \multimap D\rangle^p a & \to & \{H, b^* = D_\multimap(a)\}b^* \\ \mathit{WrapDyn}\text{ - }! & \{H\}\langle D \Leftarrow !D\rangle^p a & \to & \{H, b^* = D_!(a)\}b^* \end{array}$$

Functorial casts

$$\begin{array}{rlll} \mathit{Cast}\text{ - }D & \{H\}\langle D \Leftarrow D\rangle^p a & \to & \{H\}a \\ \mathit{Cast}\text{ - }\star & \{H\}\langle \star \Leftarrow \star\rangle^p a & \to & \{H\}a \\ \mathit{Cast}\text{ - }\otimes & \{H\}\langle t_1' \otimes t_2' \Leftarrow t_1 \otimes t_2\rangle^p a & \to & \{H\}\texttt{let } {a_1}^* \otimes {a_2}^* = a \texttt{ in} \\ & & & \qquad \langle t_1' \Leftarrow t_1\rangle^p {a_1}^* \otimes \langle t_2' \Leftarrow t_2\rangle^p {a_2}^* \\ \mathit{Cast}\text{ - }\multimap & \{H\}\langle t_1' \multimap t_2' \Leftarrow t_1 \multimap t_2\rangle^p a & \to & \{H\}\lambda b^*.\ \langle t_2' \Leftarrow t_2\rangle^p (a\ (\langle t_1 \Leftarrow t_1'\rangle^{\bar{p}} b^*)) \\ \mathit{Cast}\text{ - }! & \{H\}\langle !t' \Leftarrow !t\rangle^p a & \to & \{H\}\texttt{let } !x^* = a \texttt{ in } !(y^* = \langle t' \Leftarrow t\rangle^p x^*) \end{array}$$

Casting from D

$$\begin{array}{rlll} \mathit{FromDyn}\text{ - }\star & \{H, a = D_\star(a')\}\langle \star \Leftarrow D\rangle^p a & \to & \{H\}a' \\ \mathit{FromDyn}\text{ - }\otimes & \{H, a = D_\otimes(a')\}\langle t_1 \otimes t_2 \Leftarrow D\rangle^p a & \to & \{H\}\langle t_1 \otimes t_2 \Leftarrow D \otimes D\rangle^p a' \\ \mathit{FromDyn}\text{ - }\multimap & \{H, a = D_\multimap(a')\}\langle t_1 \multimap t_2 \Leftarrow D\rangle^p a & \to & \{H\}\langle t_1 \multimap t_2 \Leftarrow D \multimap D\rangle^p a' \\ \mathit{FromDyn}\text{ - }! & \{H, a = D_!(a')\}\langle !t \Leftarrow D\rangle^p a & \to & \{H\}\langle !t \Leftarrow !D\rangle^p a' \end{array}$$

Failing casts from D

$$\begin{array}{rlllr} \mathit{CastFail}\text{ - }\star & \{H, a = D_\star(a')\}\langle t \Leftarrow D\rangle^p a & \to & \Uparrow p & \text{if } t \neq \star \text{ and } t \neq D \\ \mathit{CastFail}\text{ - }\otimes & \{H, a = D_\otimes(a')\}\langle t \Leftarrow D\rangle^p a & \to & \Uparrow p & \text{if } t \neq t_1 \otimes t_2 \text{ and } t \neq D \\ \mathit{CastFail}\text{ - }\multimap & \{H, a = D_\multimap(a')\}\langle t \Leftarrow D\rangle^p a & \to & \Uparrow p & \text{if } t \neq t_1 \multimap t_2 \text{ and } t \neq D \\ \mathit{CastFail}\text{ - }! & \{H, a = D_!(a')\}\langle t \Leftarrow D\rangle^p a & \to & \Uparrow p & \text{if } t \neq !t' \text{ and } t \neq D \end{array}$$

Fig. 7. Reduction rule extensions.

Ground types

$$g ::= \star \mid D \otimes D \mid D \multimap D \mid !D$$

Subtyping $\boxed{t <: t'}$

$$\frac{}{\star <: \star} \quad \frac{}{D <: D} \quad \frac{t <: g}{t <: D} \quad \frac{t_1 <: t_1' \quad t_2 <: t_2'}{t_1 \otimes t_2 <: t_1' \otimes t_2'} \quad \frac{t_1' <: t_1 \quad t_2 <: t_2'}{t_1 \multimap t_2 <: t_1' \multimap t_2'} \quad \frac{t <: t'}{!t <: !t'}$$

Positive subtyping $\boxed{t <:^+ t'}$

$$\frac{}{\star <:^+ \star} \quad \frac{}{t <:^+ D} \quad \frac{t_1 <:^+ t_1' \quad t_2 <:^+ t_2'}{t_1 \otimes t_2 <:^+ t_1' \otimes t_2'} \quad \frac{t_1' <:^- t_1 \quad t_2 <:^+ t_2'}{t_1 \multimap t_2 <:^+ t_1' \multimap t_2'} \quad \frac{t <:^+ t'}{!t <:^+ !t'}$$

Negative subtyping $\boxed{t <:^- t'}$

$$\frac{}{\star <:^- \star} \quad \frac{}{D <:^- t} \quad \frac{t <:^- g}{t <:^- t'} \quad \frac{t_1 <:^- t_1' \quad t_2 <:^- t_2'}{t_1 \otimes t_2 <:^- t_1' \otimes t_2'} \quad \frac{t_1' <:^+ t_1 \quad t_2 <:^- t_2'}{t_1 \multimap t_2 <:^- t_1' \multimap t_2'} \quad \frac{t <:^- t'}{!t <:^- !t'}$$

Fig. 8. Subtyping rules.

Wadler and Findler also define a transformation to embed untyped programs into the blame calculus. For our system, an analogous transformation can be defined for a version of the untyped linear lambda calculus [3]. For details we refer to the technical report above mentioned in Section 1.

4 Subtyping

The subtyping relation, defined in Figure 8, holds between types that can be cast without raising blame. Following the subtyping relation for the blame calculus, it is split into three relations. Positive subtyping characterizes casts that never raise positive blame, as we show in Section 5. Negative subtyping characterizes casts that never raise negative blame and the plain subtyping relation characterizes casts that never raise blame at all.

Ground types, also defined in Figure 8, play a special role in the definition of subtyping because it turns out that casting into a ground type never raises blame (viz. the definition of negative subtyping). Ground types increase the scope of the subtyping relation based on this observation.

The subtyping rules involving D, the base type $\star$, and the linear function type are identical to Wadler and Findler's rules. The rules for pairs and exponentials are new but analogous to the existing rules: both are covariant and immutable. They do not give rise to new problems and similar results hold.

Lemma 3 (Factoring subtyping).
$t_1 <: t_2$ if and only if $t_1 <:^+ t_2$ and $t_1 <:^- t_2$.

The inference rules for $t' <:^+ t$ are syntax-directed which enables the proof of the following lemma.

$\boxed{e \text{ sf } p}, \boxed{s \text{ sf } p}$

$$\frac{t_2 <:^{+} t_1 \quad e \text{ sf } p}{\langle t_1 \Leftarrow t_2 \rangle^{p} e \text{ sf } p} \qquad \frac{t_2 <:^{-} t_1 \quad e \text{ sf } p}{\langle t_1 \Leftarrow t_2 \rangle^{\bar{p}} e \text{ sf } p} \qquad \frac{p \neq q \quad p \neq \bar{q} \quad e \text{ sf } p}{\langle t_1 \Leftarrow t_2 \rangle^{q} e \text{ sf } p}$$

$$\frac{}{\star \text{ sf } p} \qquad \frac{}{a \text{ sf } p} \qquad \cdots \qquad \frac{e \text{ sf } p}{\lambda a.\ e \text{ sf } p} \qquad \cdots$$

Fig. 9. Blame-safe expressions.

Lemma 4. *If* $D <:^{+} t$ *then* $t = D$.

We have not considered naive subtyping in this paper as it is not needed for proving the blame theorem.

5 The Blame Theorem

The slogan of the blame theorem [22] is that "well typed programs can't be blamed". In other words, if a program raises blame, a dynamically typed expression is responsible.

To prove this theorem, Wadler and Findler define the notion of *safe expressions*. A safe expression for a specific blame label will never raise that blame label. This notion of blame safety can only be violated through a cast expression $\langle t' \Leftarrow t \rangle^{p} e$ and it can be derived from the particular subtyping relation which holds between the types t' and t. Therefore, given preservation and progress of the reduction rules with respect to blame safety, the potential for raising blame is defined by the types used in the individual cast expressions of a program.

We replicate this reasoning for the linear blame calculus by defining blame safety and showing its preservation and progress for typed configurations. It is not sufficient to define blame safety just for expressions like in the blame calculus, because our operational semantics is defined in terms of heap-expression configurations $\{H\}e$. As variables can refer to expressions in the heap, they are only safe if the referred expressions are safe. A sufficient condition to ensure safety is therefore to require the entire heap to contain safe expressions for a particular blame label. The precise definition may be found in the technical report.

We also explored the idea of making variables safe depending on the expression that they refer to in the heap. However, this idea turned out to require a lot more formalism (e.g., formalizing reachability in the heap) because of recursive bindings in the heap.

Some of the adapted rules for blame safety of expressions are shown in Figure 9. The complete definition is given in the technical report. The cast expressions are the only places, where blame safety may be violated. The rules for cast expressions are analogous to those of the blame calculus: a cast is safe for p if it has label p and its types are related by positive subtyping, if it has label $\bar{p}$ and its

types are related by negative subtyping, or if it has a label unrelated to p. The remaining rules just propagate the safety requirement to their subexpressions. Variables, wrapped variables, and the unit value are always safe.

The preservation lemma for blame safety establishes the following fact: The reduction of a configuration which is safe for a blame label p results in a configuration which is again safe for p.

Lemma 5 (Safe configurations: preservation). *Suppose that* $\vdash H : \Theta; \Gamma$ *and* $\Theta; \Gamma \vdash e : t$ *and* $\{H\}e$ sf p *and* $\{H\}e \to \{H'\}e'$ *then* $\{H'\}e'$ sf p.

The proof is by induction on the reduction relation.

The next essential part to the blame theorem is the fact that blame-free progress of safe configurations is guaranteed:

Lemma 6 (Progress of safe configurations).
If $\{H\}e$ sf p *then* $\{H\}e \not\to \Uparrow p$.

Proof. We actually prove the contraposition "If $\{H\}e \to \Uparrow p$ then *not* e sf p" by induction on the reduction relation. This induction uses Lemma 4.

Like in Wadler and Findler's blame calculus the blame theorem is a corollary of Lemmas 5 and 6:

Corollary 1 (Well typed linear programs can't be blamed). *Let* $\{H\}e$ *be a well typed configuration and* $\langle t_1 \Leftarrow t_2\rangle^p f$ *a subexpression of* e *or a subexpression in a binding in* H *containing the only occurrence of* p *in any expression reachable by* $\{H\}e$. *If* $t_2 <:^+ t_1$ *then* $\{H\}e \not\to^* \Uparrow p$. *If* $t_2 <:^- t_1$ *then* $\{H\}e \not\to^* \Uparrow \bar{p}$. *If* $t_2 <: t_1$ *then* $\{H\}e \not\to^* \Uparrow p$ *and* $\{H\}e \not\to^* \Uparrow \bar{p}$.

6 Shortcut Casts

Consider the following simple example:

$$\texttt{let } f = \langle D \Leftarrow !(int \multimap int)\rangle !\lambda a.\ a + 1 \texttt{ in}$$
$$\texttt{let } !f_1 = \langle !D \Leftarrow D\rangle f \texttt{ in } \langle int \multimap int \Leftarrow D\rangle f_1\ 0$$

The dynamically typed variable f is used exactly once, but an extra cast and exponential elimination has to be inserted in the second line to satisfy the type checker. The problem in this case is that the variable f "hides" a value of type $!(int \multimap int)$, but the single use of the function requires the unreplicated type $(int \multimap int)$.

This mismatch can be addressed by extending our system with "subtyping" of the form $!t <: t$ where the conversion from the left side to the right side must be made explicit via casting. This notion of subtyping is based on the observation that each replicable value can serve as a linear value. In a first step, we only address casting from a dynamic type that wraps a replicated type to the underlying unreplicated type. To this end, we replace the *CastFail-!* rule by

a rule that directly dereferences the exponential and then retries the cast on the unwrapped value (after evaluation of x):

$$\mathit{FromDyn}\text{ - }!t \quad \{H, a = D_!(a'), a' = !x\}\langle t \Leftarrow D\rangle^p a \;\; \rightarrow \;\; \{H\}\langle t \Leftarrow D\rangle^p x \quad \text{if } t \neq !t'$$

With this rule in place, the example above can be rewritten as follows without explicitly casting f from D to $!D$:

$$\begin{array}{l}\texttt{let } f = \langle D \Leftarrow !(\mathit{int} \multimap \mathit{int})\rangle !\lambda a.\ a + 1 \texttt{ in} \\ \quad \langle \mathit{int} \multimap \mathit{int} \Leftarrow D\rangle f\ 0\end{array}$$

The linear f, which wraps a value of exponential type, is directly converted to a linear function. This use of f is more convenient than explicit unwrapping.

Our semantics also handles the unlikely case of a multiply wrapped type like $!!!\star$. Converting a value of this type into the dynamic type requires $\texttt{let } s = \langle D \Leftarrow !!!\star\rangle !!!\star \texttt{ in } \ldots$ whereas using it (once) just requires casting with $\langle \star \Leftarrow D\rangle s$.

All the results we have so far, type soundness, properties of subtyping, and the blame theorem, still hold without change.

A less satisfactory consequence is that some sequences of cast expressions cannot be simplified, anymore. As an example, consider the expression

$$\texttt{let } f : \mathit{int} \multimap \mathit{int} = \ldots \texttt{ in } f\ (\langle \mathit{int} \Leftarrow D\rangle(\langle D \Leftarrow !\mathit{int}\rangle !42))$$

This expression is legal in the type system of Section 3 and it executes without run-time errors in the extended semantics of the current section. In other systems, it is possible to optimize such a (non-failing) sequence of casts to just a single cast as in:

$$f\ (\langle \mathit{int} \Leftarrow !\mathit{int}\rangle !42)$$

However, our present system does not permit this reduced cast. In fact, our operational semantics cannot execute this cast because its left and right side types are not compatible. This restriction is also enforced by the type system.

This problem can be amended by adding evaluation rules for the evaluation of casts from an exponential type to a non-dynamic type. The strategy is the same as for the *FromDyn* - $!t$ rule: eliminate the exponential and retry the cast.

$$\begin{array}{rll}
\mathit{Cast}{-}!-\star & \{H, a = !x\}\langle \star \Leftarrow !t\rangle^p a & \rightarrow \; \{H\}\langle \star \Leftarrow t\rangle^p x \\
\mathit{Cast}{-}!-\otimes & \{H, a = !x\}\langle t_1 \otimes t_2 \Leftarrow !t\rangle^p a & \rightarrow \; \{H\}\langle t_1 \otimes t_2 \Leftarrow t\rangle^p x \\
\mathit{Cast}{-}!-\multimap & \{H, a = !x\}\langle t_1 \multimap t_2 \Leftarrow !t\rangle^p a & \rightarrow \; \{H\}\langle t_1 \multimap t_2 \Leftarrow t\rangle^p x
\end{array}$$

For these casts to be admissible in a program, the compatibility relation needs to be amended to reflect $!t <: t$ subtyping. By including this subtyping relation, compatibility is no longer symmetric, but it turns into a reflexive, antisymmetric relation. Also the rule for two compatible function types is now contravariant in the argument type. Compatibility is not transitive to rule out casts like $\langle \star \Leftarrow t_1 \otimes t_2\rangle$ that always fail. Compatibility ensures that casts only fail

$\boxed{t \lesssim t'}$

$$\frac{}{\star \lesssim \star} \quad \frac{}{t \lesssim D} \quad \frac{}{D \lesssim t} \quad \frac{t_1 \lesssim t_1' \quad t_2 \lesssim t_2'}{t_1 \otimes t_2 \lesssim t_1' \otimes t_2'} \quad \frac{t_1' \lesssim t_1 \quad t_2 \lesssim t_2'}{t_1 \multimap t_2 \lesssim t_1' \multimap t_2'} \quad \frac{t \lesssim t'}{!t \lesssim !t'}$$

$$\frac{t \lesssim t_1 \multimap t_2}{!t \lesssim t_1 \multimap t_2} \quad \frac{t \lesssim t_1 \otimes t_2}{!t \lesssim t_1 \otimes t_2} \quad \frac{t \lesssim \star}{!t \lesssim \star}$$

Fig. 10. Compatibility with shortcut subtyping.

when trying to unwrap a dynamic type. Figure 10 shows the updated definition of the compatibility relation. It strictly encompasses the original notion of compatibility. The typing rule for casts must be updated accordingly.

$$\frac{\Theta; \Gamma \vdash e : t_2 \quad t_2 \lesssim t_1}{\Theta; \Gamma \vdash \langle t_1 \Leftarrow t_2 \rangle^p e : t_1} \; Cast'$$

With this extended framework in place, we can include a simplification rule for casts as follows:

$$\langle t_1 \Leftarrow t_2 \rangle^p \langle t_2 \Leftarrow t_3 \rangle^q e \Longrightarrow \langle t_1 \Leftarrow t_3 \rangle^{pq} e \qquad \text{if } t_3 \lesssim t_1$$

The blame labels for the two casts have to be combined to cater for a sequence of casts like

$$\langle \mathit{int} \otimes \mathit{int} \Leftarrow \mathit{int} \otimes D \rangle^p \langle \mathit{int} \otimes D \Leftarrow D \otimes D \rangle^q e$$

where both steps involve an unwrapping of a dynamic type, each of which may have to be attributed to a different part of the program, that is, either p or q.

Theorem 2. *Type soundness holds for the extended calculus with the Cast′ rule and the modified operational semantics.*

Proof. By extending the proofs of type preservation and progress with the cases for the four modifications of the reduction rules.

Working towards a blame theorem, the subtyping relations of Figure 8 extend in a similar way as the compatibility relation. It is sufficient to add the rules that drop the exponential as long as the target type is neither dynamic nor another exponential. This choice also keeps the subtyping relations deterministic. As these rules are identical for all three relations, $<:$, $<:^+$, and $<:^-$, we only show them for one relation.

$$\frac{t <:^+ \star}{!t <:^+ \star} \qquad \frac{t <:^+ t_1' \multimap t_2'}{!t <:^+ t_1' \multimap t_2'} \qquad \frac{t <:^+ t_1' \otimes t_2'}{!t <:^+ t_1' \otimes t_2'}$$

The lemmas in Section 4 still hold for the extended subtyping relations. Also the establishment of the blame theorem and its preliminaries carry over by extending the proofs with the four cases of the modified reduction rules.

7 Related Work

We based our work on Turner and Wadler's linear lambda calculus [21] and extended it with recursion as in Lily [6]. We considered other versions of linear lambda calculus as alternative starting points (for example [2–4]). However, we chose Turner and Wadler's because its formalization of linear values as heap references makes linearity very explicit.

Affine types are closely related to linear types. An affine value must be eliminated at most once, but it may also be discarded. The results of the paper extend readily to affine systems. Again we refer to the technical report for details.

In the introduction, we refer to a number of papers on statically typed languages with type `Dynamic`. Actually, there are different flavors of such languages. The first (earlier) flavor is the one treated by Abadi and coworkers [1]. It is not concerned with type casts, but rather has a specific expression to create a dynamic value by pairing a standard value with its static type. The corresponding, type-safe elimination form is a typecase construct that performs pattern matching on the type component and extracts the value in case of a match. There is a bulk of further work in this area, which we choose not to comment on, because we do not consider typecase in this work. We conjecture —based on our results— that linearity is also orthogonal to dynamics in a language with typecase.

The flavor that we are interested in starts with Henglein's investigation of dynamic typing [10], which pioneered the ideas of wrapped values and of safe and unsafe casts in the context of a simply-typed lambda calculus. However, Henglein does not introduce a subtyping relation that would have enabled him to prove a blame theorem.

Taha and Siek [14,16] have proposed the use of subtyping to characterize the potential failure of a cast expression. Their work has been extended by Wadler and Findler with blame assignment, the marking of cast expressions with unique labels to determine the program point that caused a cast failure. On this basis, they proved the blame theorem, which states that errors in a gradually typed program are always blamed on the untyped (dynamic) part of the program.

Henglein [10] introduces the idea of a coercion calculus in order to reduce the number of times a type is tested at run time. This idea has been picked up by a number of researches with different goals: eager reporting of cast errors [15], improving the efficiency of gradual typing [11], providing streamlined data structures and algorithms for representing and normalizing coercions [17]. We only touch on this subject briefly in Section 6 to indicate that our approach is compatible with coercion calculi.

We are not aware of any work that has explored the interaction of linearity and gradual typing.

8 Conclusion

We extended a linearly typed lambda calculus with recursion and type `Dynamic`, proved type soundness for it, and established a blame theorem a la Wadler and

Findler. In comparison to Wadler and Findler's work, our calculus is more modular thus making it easy to extend with additional type constructions or with optimizations as shown in Section 6. As an extension, we integrated a notion of subtyping that allows single uses of dynamic replicated values not to require explicit unwrapping of the exponential, but rather just casting from the dynamic type to the linear target type. Thus, we have shown that linear typing and gradual typing with blame assignment are orthogonal aspects in a lambda calculus setting. We have also demonstrated the robustness of the concepts established with the blame theorem by extending the underlying calculus with a notion of subtyping. Although we have considered a core calculus, adding datatypes and conditionals would be straightforward.

There are a number of avenues for further work. It would be interesting to consider the interaction of linearity, polymorphism, and gradual typing to see if the orthogonality found in this work can be sustained. Because of the modularity of our approach, we expect the orthogonality to carry over to the context of session typing and probably also to more general process calculi. Last, instead of having the dynamic type forget the type structure, it could also forget about the linearity restriction. Similar ideas have been pursued by Pucella and Tov [19,20] and they could also be considered for session typing and process calculi.

References

1. M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *ACM TOPLAS*, 13(2):237–268, Apr. 1991.
2. S. Abramsky. Computational interpretations of linear logic. *Theor. Comput. Sci.*, 111(1&2):3–57, 1993.
3. S. Alves, M. Fernández, M. Florido, and I. Mackie. Gödel's system T revisited. *Theoretical Computer Science*, 411(11-13):1484–1500, 2010.
4. P. N. Benton, G. M. Bierman, V. de Paiva, and M. Hyland. A term calculus for intuitionistic linear logic. In M. Bezem and J. F. Groote, editors, *TLCA*, volume 664 of *LNCS*, pages 75–90. Springer, 1993.
5. G. M. Bierman, E. Meijer, and M. Torgersen. Adding dynamic types to C#. In T. D'Hondt, editor, *ECOOP*, volume 6183 of *LNCS*, pages 76–100, Maribor, Slovenia, 2010. Springer.
6. G. M. Bierman, A. M. Pitts, and C. V. Russo. Operational properties of Lily, a polymorphic linear lambda calculus with recursion. *Electr. Notes Theor. Comput. Sci.*, 41(3):70–88, 2000.
7. L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In P. Gastin and F. Laroussinie, editors, *CONCUR*, volume 6269 of *Lecture Notes in Computer Science*, pages 222–236, Paris, France, Aug. 2010. Springer.
8. G. Castagna, editor. *Proc. of the 18th ESOP*, volume 5502 of *LNCS*, York, United Kingdom, Mar. 2009. Springer.
9. J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
10. F. Henglein. Dynamic typing: Syntax and proof theory. *Science of Computer Programming*, 22:197–230, 1994.
11. D. Herman, A. Tomb, and C. Flanagan. Space-efficient gradual typing. In *Trends in Functional Programming (TFP)*, 2007.

12. J. Matthews and R. B. Findler. Operational semantics for multi-language programs. *ACM TOPLAS*, 31:12:1–12:44, Apr. 2009.
13. J. Palsberg, editor. *Proc. 37th ACM Symp. POPL*, Madrid, Spain, Jan. 2010. ACM Press.
14. J. Siek and W. Taha. Gradual typing for objects. In E. Ernst, editor, *21st ECOOP*, volume 4609 of *LNCS*, pages 2–27, Berlin, Germany, July 2007. Springer.
15. J. G. Siek, R. Garcia, and W. Taha. Exploring the design space of higher-order casts. In Castagna [8], pages 17–31.
16. J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, Sept. 2006.
17. J. G. Siek and P. Wadler. Threesomes, with and without blame. In Palsberg [13], pages 365–376.
18. K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In C. Halatsis, D. Maritsas, G. Philokyprou, and S. Theodoridis, editors, *6th International PARLE Conference (Athens, Greece, July 1994)*, volume 817 of *LNCS*, pages 398–413. Springer, 1994.
19. J. A. Tov and R. Pucella. Stateful contracts for affine types. In A. D. Gordon, editor, *ESOP 2010*, volume 6012 of *LNCS*, pages 550–569. Springer, 2010.
20. J. A. Tov and R. Pucella. Practical affine types. In *Proc. 38th ACM Symp. POPL*, pages 447–458, Austin, TX, USA, Jan. 2011. ACM Press.
21. D. N. Turner and P. Wadler. Operational interpretations of linear logic. *Theoretical Computer Science*, 227(1-2):231–248, 1999.
22. P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In Castagna [8], pages 1–16.
23. A. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.
24. T. Wrigstad, F. Z. Nardelli, S. Lebresne, J. Östlund, and J. Vitek. Integrating typed and untyped code in a scripting language. In Palsberg [13], pages 377–388.

A Untyped linear lambda calculus

The type system extensions introduced in Section 3 also enable a mixture of typed and untyped code like in the blame calculus. Figure 11 defines an untyped linear lambda calculus. The expression syntax is the same as for the typed calculus, except that it omits casts and adds the possibility to embed typed expressions with the form $\lfloor e \rfloor$.

A well-formedness relation $\Theta; \Gamma \vdash u$ wf enforces linearity by bookkeeping about variable usage with type environments that bind all variables to type D. It is closely related to the untyped linear lambda calculus [3], which just keeps track of variable occurrences. In our work, the environments are necessary to support the embedding of typed terms.

Figure 12 defines a transformation $\lceil u \rceil$ to embed untyped expressions in typed programs by inserting the required casts (analogous to the blame calculus). A similar correctness result holds (by induction on the well-formedness relation):

Lemma 7. $\Theta; \Gamma \vdash u$ wf *if and only if* $\Theta; \Gamma \vdash \lceil u \rceil : D$

Syntax

$$\begin{array}{lll} \text{untyped terms} & u ::= & x \mid a \mid \lambda a.\ u \mid u\ u \mid\ \star \mid \texttt{let}\ \star = u\ \ \texttt{in}\ u \mid \\ & & u \otimes u \mid \texttt{let}\ a \otimes a = u\ \ \texttt{in}\ u \mid\ !(x = u) \mid \texttt{let}\ !x = u\ \ \texttt{in}\ u \mid \lfloor e \rfloor \end{array}$$

Well formed terms $\boxed{\Theta; \Gamma \vdash u\ \text{ wf}}$

$$\frac{\Theta; \Gamma \vdash e : D}{\Theta; \Gamma \vdash \lfloor e \rfloor\ \text{ wf}} \qquad \frac{}{\Theta;\ \texttt{-}\ \vdash \star\ \text{ wf}} \qquad \frac{\Theta; \Gamma \vdash u\ \text{ wf} \quad \Theta; \Delta \vdash u\ \text{ wf}}{\Theta; \Gamma, \Delta \vdash \texttt{let}\ \star = u\ \ \texttt{in}\ v\ \text{ wf}}$$

$$\frac{}{\Theta; a : D \vdash a\ \text{ wf}} \qquad \frac{\Theta; \Gamma, a : D \vdash u\ \text{ wf}}{\Theta; \Gamma \vdash \lambda a.\ u\ \text{ wf}} \qquad \frac{\Theta; \Gamma \vdash u\ \text{ wf} \quad \Theta; \Gamma \vdash v\ \text{ wf}}{\Theta; \Gamma, \Delta \vdash u\ v\ \text{ wf}}$$

$$\frac{\Theta; \Gamma \vdash u\ \text{ wf} \quad \Theta; \Delta \vdash v\ \text{ wf}}{\Theta; \Gamma, \Delta \vdash u \otimes v\ \text{ wf}} \qquad \frac{\Theta; \Gamma \vdash u\ \text{ wf} \quad \Theta; \Delta, a : D, b : D \vdash v\ \text{ wf}}{\Theta; \Gamma, \Delta \vdash \texttt{let}\ a \otimes b = u\ \ \texttt{in}\ v\ \text{ wf}}$$

$$\frac{}{\Theta, x : D; \Gamma \vdash x\ \text{ wf}} \qquad \frac{\Theta, x : D;\ \texttt{-}\ \vdash u\ \text{ wf}}{\Theta;\ \texttt{-}\ \vdash !(x = u)\ \text{ wf}} \qquad \frac{\Theta, x : D; \Gamma \vdash u\ \text{ wf} \quad \Theta, x : D; \Delta \vdash v\ \text{ wf}}{\Theta; \Gamma, \Delta \vdash \texttt{let}\ !x = u\ \ \texttt{in}\ v\ \text{ wf}}$$

Fig. 11. Untyped linear lambda calculus.

Embedding $\boxed{\lceil u \rceil}$

$$\begin{array}{rcl} \lceil a \rceil & = & a \\ \lceil x \rceil & = & x \\ \lceil \star \rceil & = & \langle D \Leftarrow \star \rangle^p \star \quad p \text{ fresh} \\ \lceil \texttt{let}\ \star = u\ \ \texttt{in}\ v \rceil & = & \texttt{let}\ \star = \langle \star \Leftarrow D \rangle^p \lceil u \rceil\ \ \texttt{in}\ \lceil v \rceil \quad p \text{ fresh} \\ \lceil \lambda a.\ u \rceil & = & \langle D \Leftarrow D \multimap D \rangle^p (\lambda a. \lceil u \rceil) \quad p \text{ fresh} \\ \lceil u\ v \rceil & = & (\langle D \multimap D \Leftarrow D \rangle^p \lceil u \rceil)\ \lceil v \rceil \quad p \text{ fresh} \\ \lceil u \otimes v \rceil & = & \langle D \Leftarrow D \otimes D \rangle^p (\lceil u \rceil \otimes \lceil v \rceil) \quad p \text{ fresh} \\ \lceil \texttt{let}\ a \otimes b = u\ \ \texttt{in}\ v \rceil & = & \texttt{let}\ a \otimes b = \langle D \otimes D \Leftarrow D \rangle^p \lceil u \rceil\ \ \texttt{in}\ \lceil v \rceil \quad p \text{ fresh} \\ \lceil !(x = u) \rceil & = & \langle D \Leftarrow !D \rangle^p !(x = \lceil u \rceil) \quad p \text{ fresh} \\ \lceil \texttt{let}\ !x = u\ \ \texttt{in}\ v \rceil & = & \texttt{let}\ !x = \langle !D \Leftarrow D \rangle^p \lceil u \rceil\ \ \texttt{in}\ \lceil v \rceil \quad p \text{ fresh} \\ \lceil \lfloor e \rfloor \rceil & = & e \end{array}$$

Fig. 12. Embedding transformation.

B Expression Safety and Heap Safety

The complete definitions for expression and heap safety are given in Figure 13.

$$\frac{t_2 <:^{+} t_1 \quad e \text{ sf } p}{\langle t_1 \Leftarrow t_2 \rangle^{p} e \text{ sf } p} \qquad \frac{t_2 <:^{-} t_1 \quad e \text{ sf } p}{\langle t_1 \Leftarrow t_2 \rangle^{\bar{p}} e \text{ sf } p} \qquad \frac{p \neq q \quad p \neq \bar{q} \quad e \text{ sf } p}{\langle t_1 \Leftarrow t_2 \rangle^{q} e \text{ sf } p}$$

$$\frac{}{\star \text{ sf } p} \qquad \frac{}{a \text{ sf } p} \qquad \frac{}{x \text{ sf } p} \qquad \frac{}{D_c(a) \text{ sf } p} \qquad \frac{e \text{ sf } p}{\lambda a.\, e \text{ sf } p}$$

$$\frac{e \text{ sf } p \quad f \text{ sf } p}{e \otimes f \text{ sf } p} \qquad \frac{e \text{ sf } p}{!(x = e) \text{ sf } p} \qquad \frac{e \text{ sf } p \quad f \text{ sf } p}{\texttt{let} \star = e \ \texttt{in} \ f \text{ sf } p}$$

$$\frac{e \text{ sf } p \quad f \text{ sf } p}{e\ f \text{ sf } p} \qquad \frac{e \text{ sf } p \quad f \text{ sf } p}{\texttt{let}\ a_1 \otimes a_2 = e \ \texttt{in} \ f \text{ sf } p} \qquad \frac{e \text{ sf } p \quad f \text{ sf } p}{\texttt{let}\ !x = e \ \texttt{in} \ f \text{ sf } p}$$

$$\frac{}{\cdot \text{ sf } p} \qquad \frac{s \text{ sf } p \quad H \text{ sf } p}{H, a = s \text{ sf } p} \qquad \frac{e \text{ sf } p \quad H \text{ sf } p}{H, x = e \text{ sf } p} \qquad \frac{H \text{ sf } p \quad e \text{ sf } p}{\{H\} e \text{ sf } p}$$

Fig. 13. Blame-safe expressions $\boxed{e \text{ sf } p}$, $\boxed{s \text{ sf } p}$ and heaps $\boxed{H \text{ sf } p}$.

C Affine Systems

The results of the paper extend readily to affine systems. An affine lambda calculus requires that every restricted value is used *at most* once (whereas linearity asks for *exactly once*). Thus, considering affinity involves only a few changes to the definitions and to the technical results.

The syntax and the semantics of the affine lambda calculus are identical to the linear lambda calculus. The typing rule *(LVar)* for linear variables changes to rule *(AVar)* to reflect affinity: it admits a non-empty linear type environment.

$$\frac{}{\Theta; \Gamma, a : t \vdash a : t}\ \mathit{AVar}$$

The lemmas for type preservation and progress need to take into account that garbage may exist among the linear bindings in the heap. Fortunately, this garbage is already reflected in the heap type Γ in the type preservation and progress lemmas.

To transition the untyped calculus from Section A to an affine calculus, the only affected well-formedness rule is the one for linear variables, again:

$$\frac{}{\Theta; \Gamma, a : D \vdash a\ \text{wf}}$$

Again, its modification admits additional linear garbage bindings in the heap.

The rest of the technical results that regard subtyping, safety of configurations, and the actual blame theorem do not require any modifications.

D Proof of Lemma 6 (blame-safe progress)

Lemma 6 is established by proving its contraposition by induction on the reduction rules. Only the rules the *CastFail - c* and the *ContextFail* rule are considered, as those are the only ones that raise blame.

D.1 Case *CastFail - ⋆*

The considered rule is

$$\{H, a = D_\star(a')\}\langle t \Leftarrow D\rangle^p a \to \Uparrow p$$

The assumptions are

$$t \neq D \tag{1}$$
$$t \neq \star \tag{2}$$

As positive subtyping is syntax-directed it follows that

$$\text{If not } t_1 <:^+ t_2 \text{ then } \langle t_2 \Leftarrow t_1\rangle^p e \text{ is not } \textit{safe for } p \tag{3}$$

From (1) and lemma 4 is follows that $D <:^+ t$ does not hold so using (3) it follows that $\langle t \Leftarrow D\rangle^p a$ is not *safe for p*.

□

The cases *CastFail - ⊗*, *CastFail - ⊸* and *CastFail - !* work analogously.

D.2 Case *ContextFail*

The considered rule is

$$\frac{\{H\}e \to \Uparrow p}{\{H\}E[e] \to \Uparrow p}\ \textit{ContextFail}$$

The induction assumption can be directly applied:

$$e \text{ is not } \textit{safe for } \mathrm{p}$$

Inspecting the rules for blame-safety, it is evident, that e sf p is a requirement for deriving blame-safety for all possible contexts.

□

E Additional lemmas

This section states lemmas that are used in the proofs in the following sections. Most proofs for these lemmas are straight forward and therefore only indicated. Otherwise the proofs are given in later sections as noted.

E.1 Lemmas about heap types

Lemma 8 (Heap typing: linear). *If* $\vdash H, a = s : \Theta; \Gamma, a : t$ *then there exist* Δ *such that*
$\vdash H : \Theta; \Gamma, \Delta$ *and* $\Theta; \Delta \vdash s : t$

The proof is an induction on the derivation of $\vdash H, a = s : \Theta; \Gamma, a : t$. This is a result from the original definition of heap typing for the linear lambda calculus by Turner and Wadler [21].

Lemma 9 (Heap typing: non-linear). *If* $\vdash H, x = e : \Theta, x : t; \Gamma$ *then* $\Theta, x : t;$ - $\vdash e : t$.

The proof is an induction on the derivation of $\vdash H, x = e : \Theta, x : t; \Gamma$. It is a slight variation of the result presented in Turner and Wadler's work [21].

Lemma 10 (Heap contains linear values). *If* $\vdash H : \Theta; \Gamma, a : t$ *then there exists* s *and* Δ *such that* $H = H', a = s$ *and* $\Theta; \Delta \vdash s : t$.

The proof is an easy induction on the derivation of $\vdash H : \Theta; \Gamma, a : t$.

Lemma 11 (Heap contains non-linear values). *If* $\vdash H : \Theta, x : t; \Gamma$ *then there exists* e *such that* $H = H', x = e$.

The proof is an easy induction on the derivation of $\vdash H : \Theta, x : t; \Gamma$.

E.2 Lemmas about expression types

Lemma 12. *If* $\Theta; \Gamma \vdash e : t$ *then* e *defines the last applied rule in the derivation. In particular:*

- *If* $\Theta; \Gamma \vdash x : t$ *then* $\Theta = \Theta', x : t$ *and* $\Gamma =$ -
- *If* $\Theta; \Gamma \vdash a : t$ *then* $\Gamma = a : t$
- *If* $\Theta; \Gamma \vdash \star :$ *then* $\Gamma =$ -
- *If* $\Theta; \Gamma \vdash$ `let` $\star = e$ `in` $f : t$ *then* $\Theta; \Gamma' \vdash e : \star$ *and* $\Theta; \Delta \vdash f : t$ *and* $\Gamma = \Gamma', \Delta$
- *If* $\Theta; \Gamma \vdash e \otimes f : t$ *then* $\Theta; \Gamma' \vdash e : t_e$ *and* $\Theta; \Delta \vdash f : t_f$ *and* $t = t_e \otimes t_f$ *and* $\Gamma = \Gamma', \Delta$.
- *If* $\Theta; \Gamma \vdash$ `let` $a \otimes b = e$ `in` $f : t_f$ *then* $\Theta; \Gamma' \vdash e : t_a \otimes t_b$ *and* $\Theta; \Delta, a : t_a, b : t_b \vdash f : t_f$ *and* $\Gamma = \Gamma', \Delta$
- *If* $\Theta; \Gamma \vdash \lambda a.\ e : t$ *then* $\Theta; \Gamma, a : t_a \vdash e : t_e$ *and* $t = t_a \multimap t_b$
- *If* $\Theta; \Gamma \vdash e\ f : t_e$ *then* $\Theta; \Gamma' \vdash e : t_f \multimap t_e$ *and* $\Theta; \Delta \vdash f : t_f$ *and* $\Gamma = \Gamma', \Delta$

- *If* $\Theta; \Gamma \vdash !e : t$ *then* $\Theta;$ - $\vdash e : t_e$ *and* $t = !t_e$ *and* $\Gamma =$ -
- *If* $\Theta; \Gamma \vdash \texttt{let}\ !x = e\ \texttt{in}\ f : t_f$ *then* $\Theta; \Gamma' \vdash e : !t_e$ *and* $\Theta, x : t_e; \Delta \vdash f : t_f$ *and* $\Gamma = \Gamma', \Delta$
- *If* $\Theta; \Gamma \vdash \langle t \Leftarrow t' \rangle^p e : t$ *then* $\Theta; \Gamma \vdash e : t'$ *and* $t \sim t'$

The proof is pattern matching on the derivation rules for $\Theta; \Gamma \vdash e : t$.

Lemma 13. *The type-compatibility* $t \sim t'$ *relation is syntax-directed. In particular:*

- *If* $t_1 \otimes t_2 \sim t_1' \otimes t_2'$ *then* $t_1 \sim t_1'$ *and* $t_2 \sim t_2'$
- *If* $t_1 \multimap t_2 \sim t_1' \multimap t_2'$ *then* $t_1 \sim t_1'$ *and* $t_2 \sim t_2'$
- *If* $!t \sim !t'$ *then* $t \sim t'$

The proof is pattern matching on the derivation rules for $t \sim t'$

Lemma 14 (Type preservation under substitution (non-linear)). *If* $\Theta, \Theta', x : t; \Gamma \vdash e : t'$ *and* $\Theta';$ - $\vdash f : t$ *then* $\Theta, \Theta'; \Gamma \vdash e[x/f] : t'$.

The proof is by induction on the derivation of $\Theta, \Theta', x : t; \Gamma \vdash e : t'$.

E.3 Lemmas about blame-safety

Lemma 15 (Expression safety derivation is unique). *If* e sf p *then* e *defines the last applied rule in the derivation. In particular:*

- *If* $\lambda a.\ e$ sf p *then* e sf p
- *If* $e \otimes f$ sf p *then* e sf p *and* f sf p
- *If* $!e$ sf p *then* e sf p
- *If* $\texttt{let}\ \star = e\ \texttt{in}\ f$ sf p *then* e sf p *and* f sf p
- *If* $e\ f$ sf p *then* e sf p *and* f sf p
- *If* $\texttt{let}\ a \otimes b = e\ \texttt{in}\ f$ sf p *then* e sf p *and* f sf p
- *If* $\texttt{let}\ !x = e\ \texttt{in}\ f$ sf p *then* e sf p *and* f sf p
- *If* $\langle t' \Leftarrow t \rangle^p e$ sf p *then* e sf p *and* $t <:^+ t'$
- *If* $\langle t' \Leftarrow t \rangle^{\bar{p}} e$ sf p *then* e sf p *and* $t <:^- t'$
- *If* $\langle t' \Leftarrow t \rangle^{\bar{q}} e$ sf p, $q \neq p$ *then* e sf p *and* $t <:^- t'$

The proof is pattern matching on the derivation rules for e sf p.

Lemma 16 (Heap safety derivation is unique).

- *If* $H, a = s$ sf p *then* H sf p *and* s sf p
- *If* $H, x = e$ sf p *then* H sf p *and* e sf p

The proof is pattern matching on the derivation rules for H sf p.

Lemma 17 (Safety under substitution). *If* e sf p *and* f sf p *then* $e[a/f]$ sf p

The proof is an induction on the derivation of e sf p

E.4 Lemmas about subtyping

Lemma 18 (Positive subtyping derivation is unique). *If $t <:^+ t'$ then t and t' define the last rule applied in the derivation. In particular:*

- *If $t_1 \otimes t_2 <:^+ t'_1 \otimes t'_2$ then $t_1 <:^+ t'_1$ and $t_2 <: t'_2$*
- *If $t_1 \multimap t_2 <:^+ t'_1 \multimap t'_2$ then $t'_1 <:^- t_1$ and $t_2 <:^+ t'_2$*
- *If $!t <:^+ !t'$ then $t <:^+ t'$*

The proof is pattern matching on the derivation rules for $t <:^+ t'$.

Lemma 19 (Negative subtypes of D). *If $t <:^- D$ then $t = D$ or $t <:^- g$ for some ground-type g*

The proof is pattern matching on the derivation rules for $t <:^- D$.

Lemma 20 (Constructed negative subtypes).

1. *If $t_1 \multimap t_2 <:^- g$ then $t_1 = D$ and $t_2 <:^- D$*
2. *If $t_1 \otimes t_2 <:^- g$ then $t_1 <:^- D$ and $t_2 <:^- D$*
3. *If $!t <:^- g$ then $t <:^- D$*

The proof is an induction on the respective subtyping derivations (see section J).

Lemma 21. *If $t <:^- g$ for some ground type g, then $t <: g'$ for some ground type g' or $t = D$*

The proof is an induction on the respective subtyping derivations (see section K).

F Proof of preservation

Like Turner and Wadler, we need to strengthen the preservation statement for the induction:

Lemma 22 (Invariance). *If* $\{H\}e \to \{H'\}e'$ *and* $\vdash H : \Theta; \Gamma, \Delta$ *and* $\Theta; \Gamma \vdash e : t$ *then there exist* Θ' *and* Γ' *such that* $\vdash H' : \Theta, \Theta'; \Gamma', \Delta$ *and* $\Theta, \Theta'; \Gamma' \vdash e' : t$.

The original preservation statement is a direct consequence of the invariance statement. The proof in an induction over the derivation of $\{H\}e \to \{H'\}e'$. As the reduction rules for introduction and elimination of pairs, functions and units as well as most of the context cases are identical to those in Turner and Wadler's work, they will not be proven here. Instead the only rules considered are those for the cast expressions and for non-linear expressions and recursion. The lemma about unique derivations (12, 13) is used implicitly in the proofs.

F.1 Case *Context*, $\langle t \Leftarrow t' \rangle^p[\ ]$

The considered rule is

$$\frac{\{H\}e \to \{H'\}e'}{\{H\}\langle t_1 \Leftarrow t_2\rangle^p e \to \{H'\}\langle t_1 \Leftarrow t_2\rangle^p e'}$$

Inverting the *Cast* rule and using the assumption $\Theta; \Gamma \vdash \langle t_1 \Leftarrow t_2\rangle^p e : t_1$ yields

$$\Theta; \Gamma \vdash e : t_2 \tag{4}$$
$$t_1 \sim t_2 \tag{5}$$

With the assumption $\vdash H : \Theta; \Gamma, \Delta$ and (4) the induction assumption can be used to derive

$$\vdash H : \Theta, \Theta'; \Gamma', \Delta \tag{6}$$
$$\Theta, \Theta'; \Gamma' \vdash e' : t_2 \tag{7}$$

The heap type is given by (6) and the expression type can be derived:

$$\frac{(7) \quad (5)}{\Theta, \Theta'; \Gamma' \vdash \langle t_1 \Leftarrow t_2\rangle^p e' : t_1}\ \mathit{Cast}$$

□

F.2 Case *WrapDyn* - $\otimes$

The considered rule is

$$\{H\}\langle D \Leftarrow D \otimes D\rangle^p a \to \{H, a' = D_\otimes(a)\}a'$$

The assumptions are

$$\vdash H : \Theta; \Gamma, \Delta \tag{8}$$

$$\Theta; \Gamma \vdash \langle D \Leftarrow D \otimes D \rangle^p a : D \tag{9}$$

By inverting the type rule *Cast* it follows with assumption (9) that

$$\Theta; \Gamma \vdash a : D \otimes D$$

so

$$\Gamma = a : D \otimes D$$

The assumption about the heap type can be therefore rewritten:

$$\vdash H : \Theta; a : D \otimes D, \Delta \tag{10}$$

A heap type can be derived:

$$\frac{(10) \quad \dfrac{}{\Theta; a : D \otimes D \vdash D_{\otimes}(a) :}\ \mathit{DynVal}\text{ - }\otimes}{\vdash H, a' = D_{\otimes}(a) : \Theta; \Delta, a' : D}\ \mathit{Value}$$

And also an expression type:

$$\frac{}{\Theta; a' : D \vdash a' : D :}\ \mathit{Var}$$

□

The cases *WrapDyn* - $\star$, *WrapDyn* - $\multimap$ and *WrapDyn* - ! work similarly.

F.3 Case *CastDyn* - $\otimes$

The considered rule is

$$\{H\}\langle D \Leftarrow t_1 \otimes t_2 \rangle^p a \to \{H\}\langle D \Leftarrow D \otimes D \rangle^p \langle D \otimes D \Leftarrow t_1 \otimes t_2 \rangle^p a$$

The assumptions are

$$\vdash H : \Theta; a : t_1 \otimes t_2, \Delta \tag{11}$$

$$\Theta; a : t_1 \otimes t_2 \vdash \langle D \Leftarrow t_1 \otimes t_2 \rangle^p a : D \tag{12}$$

By inverting the type rule *Cast* it follows from (12) that

$$\Theta; a : t_1 \otimes t_2 \vdash a : t_1 \otimes t_2 \tag{13}$$

The heap type is directly given by (11). The expression type can be derived:

$$\frac{\dfrac{(13) \quad \dfrac{\overline{D \sim t_1} \quad \overline{D \sim t_2}}{D \otimes D \sim t_1 \otimes t_2}}{\Theta; a : t_1 \otimes t_2 \vdash \langle D \otimes D \Leftarrow t_1 \otimes t_2 \rangle^p a : D \otimes D}\ \mathit{Cast} \quad \dfrac{}{D \sim D \otimes D}}{\Theta; a : t_1 \otimes t_2 \vdash \langle D \Leftarrow D \otimes D \rangle^p \langle D \otimes D \Leftarrow t_1 \otimes t_2 \rangle^p a : D}\ \mathit{Cast}$$

□

The cases *CastDyn* - $\multimap$, *CastDyn* - ! work similarly.

F.4 Case *FromDyn* - $\otimes$

The considered rule is

$$\{H, a = D_{\otimes}(a')\}\langle t_1 \otimes t_2 \Leftarrow D\rangle^p a \to \{H\}\langle t_1 \otimes t_2 \Leftarrow D \otimes D\rangle^p a'$$

The assumptions are

$$\vdash H, a = D_{\otimes}(a') : \Theta; a : D, \Delta \tag{14}$$
$$\Theta; a : D \vdash \langle t_1 \otimes t_2 \Leftarrow D\rangle^p a : t_1 \otimes t_2 \tag{15}$$

By inverting the type rule *Cast* it follows from (15) that

$$\Theta; a : D \vdash a : D \tag{16}$$

Using the lemma 8 with (14) yields

$$\vdash H : \Theta; \Delta, \Gamma' \tag{17}$$
$$\Theta; \Gamma' \vdash D_{\otimes}(a') : D \tag{18}$$

It follows from (18) and inverting the type rule *DynVal* - $\otimes$ that

$$\Gamma' = a' : D \otimes D$$

so the expression type can be derived:

$$\frac{\Theta; a' : D \otimes D \vdash a' : D \otimes D \quad \dfrac{t_1 \sim D \quad t_2 \sim D}{t_1 \otimes t_2 \sim D \otimes D}}{\Theta; a' : D \otimes D \vdash \langle t_1 \otimes t_2 \Leftarrow D \otimes D\rangle^p a' : t_1 \otimes t_2}$$

The heap type is given by (17).

□

The cases *FromDyn* - $\star$, *FromDyn* - $\multimap$ and *FromDyn* - ! work similarly.

F.5 Case *Cast* - $\multimap$

The considered rule is

$$\{H\}\langle t_1' \multimap t_2' \Leftarrow t_1 \multimap t_2\rangle^p a \to \{H\}\lambda a'.\ \langle t_2' \Leftarrow t_2\rangle^p (a\ (\langle t_1 \Leftarrow t_1'\rangle^{\bar{p}} a'))$$

The assumptions are

$$\vdash H : \Theta; a : t_1 \multimap t_2, \Delta \tag{19}$$
$$\Theta; a : t_1' \multimap t_2' \vdash \langle t_1' \multimap t_2' \Leftarrow t_1 \multimap t_2\rangle^p a : t_1' \multimap t_2' \tag{20}$$

From (20) it follows that $t_1' \multimap t_2' \sim t_1 \multimap t_2$. This yields

$$t_1' \sim t_1 \tag{21}$$
$$t_2' \sim t_2 \tag{22}$$

An expression type can directly be derived:

$$\dfrac{\dfrac{\dfrac{\Theta; a : t_1 \multimap t_2 \vdash a : t_1 \multimap t_2 \quad \dfrac{\dfrac{}{\Theta; a' : t_1' \vdash a' : t_1'}\ LVar \quad (21)}{\Theta; a' : t_1' \vdash \langle t_1 \Leftarrow t_1' \rangle^{\bar{p}} a' : t_1}\ Cast}{\Theta; a : t_1 \multimap t_2, a' : t_1' \vdash (a\ (\langle t_1 \Leftarrow t_1' \rangle^{\bar{p}} a') : t_2}\ \multimap\text{-}E \quad (22)}{\Theta; a : t_1 \multimap t_2, a' : t_1' \vdash \langle t_2' \Leftarrow t_2 \rangle^{p} (a\ (\langle t_1 \Leftarrow t_1' \rangle^{\bar{p}} a') : t_2'}\ Cast}{\Theta; a : t_1 \multimap t_2 \vdash \lambda a'.\ \langle t_2' \Leftarrow t_2 \rangle^{p} (a\ (\langle t_1 \Leftarrow t_1' \rangle^{\bar{p}} a')) : t_1' \multimap t_2'}\ \multimap\text{-}I$$

The heap type is given by (19).

□

The cases *Cast* - $\otimes$, *Cast* - !, *Cast* - $\star$ and *Cast* - *D* work similarly.

F.6 Case !-*I*

The considered rule is

$$\{H\}!(x = e) \to \{H, b =!y, y = e[x/y]\}b$$

The assumptions in this case are

$$\vdash H : \Theta; \Gamma, \Delta \tag{23}$$
$$\Theta;\ \text{-}\ \vdash !(x = e) :!t \tag{24}$$

Inversion of assumption (24) yields

$$\Theta, x : t;\ \text{-}\ \vdash e : t$$

Using this with lemma 14 and the axiom $y : t;\ \text{-}\ \vdash y : t$ yields

$$\Theta, y : t;\ \text{-}\ \vdash e[x/y] : t \tag{25}$$

Now the heap type can be derived:

$$\dfrac{\dfrac{(23) \quad (25)}{\vdash H, y = e[x/y] : \Theta, y : t; \Gamma, \Delta}\ Closure \quad \dfrac{}{\Theta, y : t; b :!t \vdash b :!t}\ \begin{matrix} LVar \\ Value \end{matrix}}{\vdash H, y = e[x/y], b =!y : \Theta, y : t; \Gamma, b :!t, \Delta}$$

The expressions typing is immediatly given by rule *LVar*:

$$\Theta, y : t; b :!t \vdash b :!t$$

F.7 Case !-*E*

The considered rule is

$$\{H, b = !y\}\texttt{let } !x = b \texttt{ in } e \to \{H\}e[x/y]$$

The assumptions in this case are

$$\vdash H, b = !y : \Theta; \Gamma, b : !t, \Delta \tag{26}$$
$$\Theta; \Gamma, b : !t \vdash \texttt{let } !x = b \texttt{ in } e : t' \tag{27}$$

Using assumption (26) with lemma 8 yields that there exists Γ' such that

$$\vdash H : \Theta; \Gamma, \Delta, \Gamma' \tag{28}$$
$$\Theta; \Gamma' \vdash !y : !t \tag{29}$$

It follows from result (29) that

$$\Gamma' = \text{ - } \tag{30}$$
$$\Theta = \Theta', y : t \tag{31}$$

The desired heap typing is given by result (28). It follows from assumption (27) and results (29) (31) that

$$\Theta', y : t, x : t; \Gamma \vdash e : t'$$

and therefore with lemma 14 that

$$\Theta', y : t; \Gamma \vdash e[x/y] : t'$$

which is the desired expression typing.

F.8 Case *Var*

The considered rule is

$$\{H, x = e\}x \to \{H, x = e\}e$$

The assumptions in this case are

$$\vdash H, x = e : \Theta, x : t; \Delta \tag{32}$$
$$\Theta, x : t; x \vdash t : \tag{33}$$

The heap typing is directly given by assumption (32). It follows from assumption (32) and lemma 9 that

$$\Theta, x : t; \text{ - } \vdash e : t$$

which is the desired expression typing.

G Proof of type progress (Lemma 2)

The proof is an induction on the derivation of $\Theta; \Gamma \vdash e : t$.

G.1 Case *Var*

The assumptions are

$$\Theta', x : t; \Gamma \vdash x : t \tag{34}$$
$$\vdash H : \Theta', x : t; \Gamma, \Delta \tag{35}$$

Using (35) with lemma 11 yields

$$H = H', x = e$$

Thus, the reduction rule *Var* can be applied.

□

G.2 Case *LVar*

In this case $e = a$ so the lemma is trivially satisfied.

G.3 Case $\star$-*I*

In this case $e = \star$, so the reduction rule $\star$-*I* can be applied

□

The cases $\multimap$ -*I* and !-*I* work similarly.

G.4 Case $\otimes$-*I*

The assumptions are

$$\Theta; \Gamma \vdash e_1 : t_1 \tag{36}$$
$$\Theta; \Delta \vdash e_2 : t_2 \tag{37}$$
$$\vdash \Theta : \Theta; \Gamma, \Delta, \Delta' \tag{38}$$

Using (36) and (38) with the induction assumption yields one of the following cases

1. there exist H' and e_1' such that $\{H\}e_1 \to \{H'\}e_1'$
2. $\{H\}e_1 \to \Uparrow p$, or
3. e_1 is a linear variable a

Case 1 Apply reduction rule *Context*.

□

Case 2 Apply reduction rule *ContextFail*.

□

Case 3 Using (37) and (38) with the induction assumption yields one of the following cases

1. there exist H' and e_2' such that $\{H\}e_2 \to \{H'\}e_2'$
2. $\{H\}e_2 \to \Uparrow p$, or
3. e_2 is a linear variable b

Case 3.1 Apply reduction rule *Context*.

□

Case 3.2 Apply reduction rule *ContextFail*.

□

Case 3.3 Reduction rule $\otimes$-I can be applied

□

G.5 Case $\otimes$-E

The assumptions are

$$\Theta; \Gamma \vdash e : t_a \otimes t_b \tag{39}$$
$$\Theta; \Delta \vdash f : t_f \tag{40}$$
$$\vdash H : \Theta; \Gamma, \Delta, \Delta' \tag{41}$$

Using (39) and (41) with the induction assumption yields one of the following cases

1. there exist H' and e' such that $\{H\}e \to \{H'\}e'$
2. $\{H\}e \to \Uparrow p$, or
3. e is a linear variable a''

Case 1 Apply reduction rule *Context*.

□

Case 2 Apply reduction rule *ContextFail*.

□

Case 3 From (39) and the fact that $e = a''$ in this case it follows that $\Gamma = a'' : t_a \otimes t_b$. Using this with lemma 10 and (41) yields

$$H = H', a'' = s$$

and

$$\Theta; \Delta'' \vdash s : t_a \otimes t_b$$

By the syntactic definition of s it is evident that it has to have the form

$$s = a' \otimes b'$$

to have type $t_a \otimes t_b$. Consequently the reduction rule $\otimes$-E can be applied.

□

The cases $\multimap$ -E, !-E, and $\star$-E work similarly.

G.6 Case *Cast*

The assumptions are

$$\Theta; \Gamma \vdash \langle t \Leftarrow t' \rangle^p e : t \tag{42}$$
$$\Theta; \Gamma \vdash e : t' \tag{43}$$
$$\vdash H : \Theta; \Gamma, \Delta' \tag{44}$$
$$t' \sim t \tag{45}$$

Using (43) and (44) with the induction assumption yields one of the following cases

1. there exist H' and e' such that $\{H\}e \to \{H'\}e'$
2. $\{H\}e \to \Uparrow p$, or
3. e is a linear variable a

Case 1 Apply reduction rule *Context*.

□

Case 2 Apply reduction rule *ContextFail*.

□

Case 3 Make a case distinction on the form of t' and t where $t \sim t'$:

Case $t = D$ One of the *CastDyn* - c, *WrapDyn* - c or the *Cast* - D rules can be used.

Case $t \neq D$ *and* $t' = D$ It follows from (43) and the fact that $e = a$ that $\Gamma = a : t'$. So, using lemma 10 with (44) yields

$$H = H', a = s$$

and

$$\Theta; \Delta'' \vdash s : D$$

By the syntactic definition of s it is evident that it has to have the form

$$s = D_c(a')$$

for some type constructor c. Thus one of the rules *UnwrapDyn* - c, *FromDyn* - c and or *CastFail* - c can be applied.

Case $t \neq D$ *and* $t' \neq D$ Either *Cast* - $\star$, *Cast* - $\otimes$, *Cast* - $\multimap$ or *Cast* - ! can be applied.

□

H Proof of safety preservation

Lemma 5 is proven by induction on the reduction rules $\{H\}e \to \{H'\}e'$. The lemmas about unique derivations (15, 16, 12, 18) are used implicitly in the proofs.

H.1 Case *Context*

Case $[\,] \otimes e$ The considered rule is

$$\frac{\{H\}e \to \{H'\}e'}{\{H\}e \otimes f \to \{H'\}e' \otimes f}$$

The assumptions are

$$\{H\}e \to \{H'\}e' \tag{46}$$
$$\vdash H : \Theta; \Gamma', \Delta', \Delta \tag{47}$$
$$\Theta; \Gamma', \Delta' \vdash e \otimes f : t_e \otimes t_f \tag{48}$$
$$e \otimes f \ \text{ sf } \ p \tag{49}$$
$$H \ \text{ sf } \ p \tag{50}$$

It follows from (49)

$$e \ \text{ sf } \ p \tag{51}$$
$$f \ \text{ sf } \ p \tag{52}$$

It follows from (48) that

$$\Theta; \Gamma' \vdash e : t_e \tag{53}$$

With (51), (47), (50) and (53) the inductions assumption applies:

$$e' \ \text{ sf } \ p \tag{54}$$
$$H' \ \text{ sf } \ p \tag{55}$$

Blame-safety for the heap is given by (55). The blame-safety for the expression can be derived:

$$\frac{(54) \quad (52)}{e \otimes f \ \text{ sf } \ p}$$

□

The *Context* cases $a \otimes [\,]$, $[\,]\ e$, $a\ [\,]$, $\texttt{let}\ a \otimes b = [\,]\ \texttt{in}\ e$, $\texttt{let}\ !x = [\,]\ \texttt{in}\ e$, $\texttt{let}\ \star = [\,]\ \texttt{in}\ e$ work similarly.

Case $\langle t_1 \Leftarrow t_2 \rangle^p[\ ]$ The considered rule is

$$\frac{\{H\}e \to \{H'\}e'}{\{H\}\langle t_1 \Leftarrow t_2\rangle^p e \to \{H'\}\langle t_1 \Leftarrow t_2\rangle^p e'}$$

The assumptions are

$$\vdash H : \Theta; \Gamma, \Delta \tag{56}$$
$$\Theta; \Gamma \vdash \langle t_1 \Leftarrow t_2\rangle^p e : t_1 \tag{57}$$
$$\langle t_1 \Leftarrow t_2\rangle^p e \text{ sf } p \tag{58}$$
$$H \text{ sf } p \tag{59}$$

It follows from (58) that

$$e \text{ sf } p \tag{60}$$
$$t_2 <:^+ t_1 \tag{61}$$

It follows from (57) that

$$\Theta; \Gamma \vdash e : t_2 \tag{62}$$
$$t_1 \sim t_2 \tag{63}$$

Using (56), (62), (60) and (59) the induction assumption applies:

$$e' \text{ sf } p \tag{64}$$
$$H' \text{ sf } p \tag{65}$$

Blame-safety for the heap is given by (65). The blame-safety for the expression can be derived:

$$\frac{(61) \quad (64)}{\langle t_1 \Leftarrow t_2\rangle^p e' \text{ sf } p}$$

□

The context cases $\langle t_1 \Leftarrow t_2\rangle^{\bar{p}}[\ ]$ and $\langle t_1 \Leftarrow t_2\rangle^q[\ ],\ q \neq p$ work similarly.

H.2 Case $\multimap$ *-I*

The considered rule is

$$\{H\}\lambda a.\ e \to \{H, b = \lambda a.\ e\}b \quad b \text{ fresh}$$

The assumptions are

$$\vdash H : \Theta; \Gamma, \Delta \tag{66}$$
$$\Theta; \Gamma \vdash \lambda a.\ e : t_a \multimap t_e \tag{67}$$
$$\lambda a.\ e \text{ sf } p \tag{68}$$
$$H \text{ sf } p \tag{69}$$

The blame-safety for the expression is immediate:

$$\frac{}{b \text{ sf } p}$$

And the blame-safety for the heap can be derived:

$$\frac{(68) \quad (69)}{H, b = \lambda a.\ e \text{ sf } p}$$

□

The cases $\otimes$-*I* and !-*I* work similarly.

H.3 Case $\multimap$ -*E*

The considered rule is

$$\{H, b = \lambda a.\ e\} b\ a' \to \{H\} e[a/a']$$

The assumptions are

$$\vdash H, b = \lambda a.\ e : \Theta; b : t_a \multimap t_e, a' : t_a, \Delta \tag{70}$$
$$\Theta; b : t_a \multimap t_e, a' : t_a \vdash b\ a' : t_a \multimap t_e \tag{71}$$
$$b\ a' \text{ sf } p \tag{72}$$
$$H, b = \lambda a.\ e \text{ sf } p \tag{73}$$

It follows from (72) that

$$b \text{ sf } p \tag{74}$$
$$a' \text{ sf } p \tag{75}$$

It follows from (73) that

$$\lambda a.\ e \text{ sf } p \tag{76}$$
$$H \text{ sf } p \tag{77}$$

Result (76) yields

$$e \text{ sf } p$$

Finally, using this with lemma 17 yields blame-safety for the expression:

$$e[a/a'] \text{ sf } p$$

Blame-safety for the heap is given by (77).

□

The cases $\otimes$-*E*, *Var* and !-*E* work similarly.

H.4 Case *CastDyn* - $\multimap$

Case $\langle D \Leftarrow t_1 \multimap t_2\rangle^p a$ The considered rule is

$$\{H\}\langle D \Leftarrow t_1 \multimap t_2\rangle^p a \to \{H\}\langle D \Leftarrow D \multimap D\rangle^p \langle D \multimap D \Leftarrow t_1 \multimap t_2\rangle^p a$$

The assumptions are

$$\vdash H : \Theta; a : t_1 \multimap t_2, \Delta \tag{78}$$
$$\Theta; a : t_1 \multimap t_2 \vdash \langle D \Leftarrow t_1 \multimap t_2\rangle^p a : D \tag{79}$$
$$\langle D \Leftarrow t_1 \multimap t_2\rangle^p a \text{ sf } p \tag{80}$$
$$H \text{ sf } p \tag{81}$$

The blame-safety for the expression can be derived:

$$\dfrac{\dfrac{}{D \multimap D <:^+ D} \quad \dfrac{\dfrac{\dfrac{}{D <:^- t_2} \quad \dfrac{}{t_1 <:^+ D}}{t_1 \multimap t_2 <:^+ D \multimap D} \quad \dfrac{}{a \text{ sf } p}}{\langle D \multimap D \Leftarrow t_1 \multimap t_2\rangle^p a \text{ sf } p}}{\langle D \Leftarrow D \multimap D\rangle^p \langle D \multimap D \Leftarrow t_1 \multimap t_2\rangle^p a \text{ sf } p}$$

The blame-safety for the heap is given by (81).

□

The case for $\langle D \Leftarrow t_1 \multimap t_2\rangle^q$ where $q \neq p$ works similarly.

Case $\langle D \Leftarrow t_1 \multimap t_2\rangle^{\bar{p}} a$ The considered rule is

$$\{H\}\langle D \Leftarrow t_1 \multimap t_2\rangle^{\bar{p}} a \to \{H\}\langle D \Leftarrow D \multimap D\rangle^{\bar{p}} \langle D \multimap D \Leftarrow t_1 \multimap t_2\rangle^{\bar{p}} a$$

The assumptions are

$$\vdash H : \Theta; a : t_1 \multimap t_2, \Delta \tag{82}$$
$$\Theta; a : t_1 \multimap t_2 \vdash \langle D \Leftarrow t_1 \multimap t_2\rangle^{\bar{p}} a : D \tag{83}$$
$$\langle D \Leftarrow t_1 \multimap t_2\rangle^{\bar{p}} a \text{ sf } p \tag{84}$$
$$H \text{ sf } p \tag{85}$$

It follows from (84) that

$$t_1 \multimap t_2 <:^- D$$

The only way to derive this negative subtype relation is by the rule

$$\frac{t <:^- g}{t <:^- t'}$$

This implies that

$$t_1 \multimap t_2 <:^- t_g \tag{86}$$

for a ground type t_g. The blame-safety for the expression can be derived:

$$\dfrac{\dfrac{\dfrac{\overline{D <:^- D} \quad \overline{D <:^+ D}}{D \multimap D <:^- D \multimap D}}{D \multimap D <:^- D} \quad \dfrac{\dfrac{(86)}{t_1 \multimap t_2 <:^- D \multimap D} \quad \overline{a \text{ sf } p}}{\langle D \multimap D \Leftarrow t_1 \multimap t_2\rangle^{\bar{p}} a \text{ sf } p}}{\langle D \Leftarrow D \multimap D\rangle^{\bar{p}} \langle D \multimap D \Leftarrow t_1 \multimap t_2\rangle^{\bar{p}} a \text{ sf } p}$$

The blame-safety for the heap is given by (85).

□

The cases *CastDyn* - $\otimes$ and *CastDyn* - ! work similarly.

H.5 Case *WrapDyn* - $\otimes$

Case $\langle D \Leftarrow D \otimes D\rangle^p a$ The considered rule is

$$\{H\}\langle D \Leftarrow t_1 \otimes t_2\rangle^p a \to \{H, a' = D_\otimes(a)\}a' \quad a' \text{ fresh}$$

The assumptions are

$$\vdash H : \Theta; a : t_1 \otimes t_2, \Delta \tag{87}$$
$$\Theta; a : t_1 \otimes t_2 \vdash \langle D \Leftarrow t_1 \otimes t_2\rangle^p a : D \tag{88}$$
$$\langle D \Leftarrow t_1 \otimes t_2\rangle^p a \text{ sf } p \tag{89}$$
$$H \text{ sf } p \tag{90}$$

The blame-safety for the expression can be derived immediately:

$$\overline{a' \text{ sf } p}$$

The blame-safety for the heap can be derived:

$$\dfrac{\overline{D_\otimes(a) \text{ sf } p} \quad (90)}{H, a' = D_\otimes(a) \text{ sf } p}$$

□

The cases $\langle D \Leftarrow D \otimes D\rangle^{\bar{p}} a$ and $\langle D \Leftarrow D \otimes D\rangle^q a$ where $q \neq p$ work analogously.

The cases *WrapDyn* - $\multimap$, *WrapDyn* - ! and *WrapDyn* - $\star$ work analogously.

H.6 Case *Cast* - $\star$

The considered rule is

$$\{H\}\langle \star \Leftarrow \star\rangle^q a \to \{H\}a$$

The assumptions are

$$H \text{ sf } p$$

which directly yield blame safety for the heap. The blame safety for the expression is also trivial:

$$\frac{}{a \text{ sf } p}$$

□

The case *Cast - D* works analogously.

H.7 Case *Cast* - $\multimap$

Case $\langle t_1' \multimap t_2' \Leftarrow t_1 \multimap t_2 \rangle^p a$ The considered rule is

$$\{H\}\langle t_1' \multimap t_2' \Leftarrow t_1 \multimap t_2\rangle^p a \to \{H\}\lambda a'.\ \langle t_2' \Leftarrow t_2\rangle^p (a\ (\langle t_1 \Leftarrow t_1'\rangle^{\bar{p}} a')) \qquad a' \text{ fresh}$$

The assumptions are

$$\vdash H : \Theta; a : t_1 \multimap t_2, \Delta \tag{91}$$
$$\Theta; a : t_1 \multimap t_2 \vdash \langle t_1' \multimap t_2' \Leftarrow t_1 \multimap t_2\rangle^p a : t_1' \multimap t_2' \tag{92}$$
$$\langle t_1' \multimap t_2' \Leftarrow t_1 \multimap t_2\rangle^p a \text{ sf } p \tag{93}$$
$$H \text{ sf } p \tag{94}$$

It follows from (93) that

$$a \text{ sf } p \tag{95}$$
$$t_1 \multimap t_2 <:^+ t_1' \multimap t_2' \tag{96}$$

From (96) it follows that

$$t_1' <:^- t_1 \tag{97}$$
$$t_2 <:^+ t_2' \tag{98}$$

The blame-safety for the expression can be derived:

$$\dfrac{\dfrac{(98)\quad \dfrac{(95)\quad \dfrac{(97)\quad \overline{a' \text{ sf } p}}{\langle t_1 \Leftarrow t_1'\rangle^{\bar{p}} a' \text{ sf } p}}{a\ (\langle t_1 \Leftarrow t_1'\rangle^{\bar{p}} a') \text{ sf } p}}{\langle t_2' \Leftarrow t_2\rangle^p (a\ (\langle t_1 \Leftarrow t_1'\rangle^{\bar{p}} a')) \text{ sf } p}}{\lambda a'.\ \langle t_2' \Leftarrow t_2\rangle^p (a\ (\langle t_1 \Leftarrow t_1'\rangle^{\bar{p}} a')) \text{ sf } p}$$

The blame-safety for the heap is given by (94)

□

Case $\langle t_1' \multimap t_2' \Leftarrow t_1 \multimap t_2\rangle^{\bar{p}} a$ The considered rule is

$$\{H\}\langle t_1' \multimap t_2' \Leftarrow t_1 \multimap t_2\rangle^{\bar{p}} a \to \{H\}\lambda a'.\ \langle t_2' \Leftarrow t_2\rangle^{\bar{p}}(a\ (\langle t_1 \Leftarrow t_1'\rangle^{p} a')) \qquad a' \text{ fresh}$$

The assumptions are

$$\langle t_1' \multimap t_2' \Leftarrow t_1 \multimap t_2\rangle^{\bar{p}} a \ \text{ sf } \ p \tag{99}$$

$$H \ \text{ sf } \ p \tag{100}$$

It follows from (99) that

$$t_1 \multimap t_2 <:^{-} t_1' \multimap t_2' \tag{101}$$

$$a \ \text{ sf } \ p \tag{102}$$

$$\tag{103}$$

Because of its syntactic form, result (101) could only be derived by two rules:

$$\frac{t <:^{-} g}{t <:^{-} t'} \tag{104}$$

or

$$\frac{t_1' <:^{+} t_1 \quad t_2 <:^{-} t_2'}{t_1 \multimap t_2 <:^{-} t_1' \multimap t_2'} \tag{105}$$

In both cases, blame safety for the heap is given by (100).

Case (104) The rule requires $t_1 \multimap t_2 <:^{-} g$ for some ground-type g. By lemma 20 it follows that

$$t_1 = D \tag{106}$$

$$t_2 <:^{-} D \tag{107}$$

From (106) it follows that

$$t_1' <:^{+} t_1 \tag{108}$$

From (107) it follows with lemma 19 that $t_2 = D$ or $t_2 <:^{-} g'$ for some ground-type g'. In both cases we have

$$t_2 <:^{-} t_2' \tag{109}$$

by

$$\frac{}{D <:^{-} t_2'} \qquad \text{or} \qquad \frac{t_2 <:^{-} g'}{t_2 <:^{-} t_2'}$$

The blame safety for the expression can now be derived:

$$\dfrac{\dfrac{\dfrac{\dfrac{(109)\quad \dfrac{(102)\quad \dfrac{(108)\quad \overline{a' \ \text{sf} \ p}}{\langle t_1 \Leftarrow t_1'\rangle^{p} a' \ \text{sf} \ p}}{a\ (\langle t_1 \Leftarrow t_1'\rangle^{p} a') \ \text{sf} \ p}}{\langle t_2' \Leftarrow t_1\rangle^{\bar{p}}(a\ (\langle t_2 \Leftarrow t_1'\rangle^{p} a')) \ \text{sf} \ p}}{\lambda a'.\ \langle t_2' \Leftarrow t_2\rangle^{\bar{p}}(a\ (\langle t_1 \Leftarrow t_1'\rangle^{p} a')) \ \text{sf} \ p}$$

Case (105) The assumptions of this rule are

$$t_1' <:^+ t_1 \tag{110}$$

$$t_2 <:^- t_2' \tag{111}$$

From this the blame safety for the expression can be derived:

$$\dfrac{\dfrac{\dfrac{(111)\quad a\ (\langle t_1 \Leftarrow t_1'\rangle^p a')\ \text{sf}\ p}{\langle t_2' \Leftarrow t_2\rangle^{\bar{p}}(a\ (\langle t_1 \Leftarrow t_1'\rangle^p a'))\ \text{sf}\ p}}{\lambda a'.\ \langle t_2' \Leftarrow t_2\rangle^{\bar{p}}(a\ (\langle t_1 \Leftarrow t_1'\rangle^p a'))\ \text{sf}\ p}}{\dfrac{(95)\quad \langle t_1 \Leftarrow t_1'\rangle^p a'\ \text{sf}\ p}{a\ (\langle t_1 \Leftarrow t_1'\rangle^p a')\ \text{sf}\ p}}\quad \dfrac{(110)\quad a'\ \text{sf}\ p}{\langle t_1 \Leftarrow t_1'\rangle^p a'\ \text{sf}\ p}$$

□

The cases *Cast* - $\otimes$ and *Cast* - ! work similarly.

H.8 Cast *FromDyn* - $\multimap$

Case $\langle t_1 \multimap t_2 \Leftarrow D\rangle^p a$ The considered rule is

$$\{H, a = D_{\multimap}(a')\}\langle t_1 \multimap t_2 \Leftarrow D\rangle^p a \to \{H\}\langle t_1 \multimap t_2 \Leftarrow D \multimap D\rangle^p a'$$

The assumptions are

$$\langle t_1 \multimap t_2 \Leftarrow D\rangle^p a\ \text{sf}\ p \tag{112}$$

from which it follows that

$$D <:^+ t_1 \multimap t_2$$

By lemma 4, this is not satisfiable. Thus this case trivially satisfies the lemma, as it reduces an unsafe expression.

□

Case $\langle t_1 \multimap t_2 \Leftarrow D\rangle^{\bar{p}} a$ The considered rule is

$$\{H, a = D_{\multimap}(a')\}\langle t_1 \multimap t_2 \Leftarrow D\rangle^{\bar{p}} a \to \{H\}\langle t_1 \multimap t_2 \Leftarrow D \multimap D\rangle^{\bar{p}} a'$$

The assumptions are

$$H, a = D_{\multimap}(a')\ \text{sf}\ p \tag{113}$$

It follows by inversion of heap blame safety that $H\ \text{sf}\ p$ which yields the blame safety of the heap directly. The blame safety of the expression can also be derived:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{D <:^+ D}\quad \dfrac{}{D <:^- D}}{D \multimap D <:^- D \multimap D}}{D \multimap D <:^- t_1 \multimap t_2}\quad \dfrac{}{a\ \text{sf}\ p}}{\langle t_1 \multimap t_2 \Leftarrow D \multimap D\rangle^{\bar{p}} a'\ \text{sf}\ p}$$

□

Case $\langle t_1 \multimap t_2 \Leftarrow D\rangle^q a \quad , q \neq p \wedge q \neq \bar{p}$ The considered rule is

$$\{H, a = D_{\multimap}(a')\}\langle t_1 \multimap t_2 \Leftarrow D\rangle^q a \to \{H\}\langle t_1 \multimap t_2 \Leftarrow D \multimap D\rangle^q a'$$

The assumptions are

$$H, a = D_{\multimap}(a') \text{ sf } p \tag{114}$$

$$q \neq p \wedge q \neq \bar{p} \tag{115}$$

It follows by inversion of heap blame safety that H sf p which yields the blame safety of the heap directly. The blame safety of the expression can also be derived:

$$\frac{(115) \quad \overline{a \text{ sf } p}}{\langle t_1 \multimap t_2 \Leftarrow D \multimap D\rangle^q \langle D \multimap D \Leftarrow D\rangle^q a \text{ sf } p}$$

□

The cases *FromDyn* - $\otimes$, *FromDyn* - $\star$ and *FromDyn* - ! work similarly.

I Proof of subtype factoring (Lemma 3)

The proof are inductions on the subtype derivations. The lemmas about unique derivations (18) are used implicitly in the proofs.

I.1 If $t <:^+ t'$ and $t <:^- t'$ then $t <: t'$

This direction is proven by an induction on the derivation of $t <:^+ t'$.

Case $\star <:^+ \star$ Immediately the subtype relation can be derived:

$$\frac{}{\star <: \star}$$

□

Case $t <:^+ D$ The assumptions are

$$t <:^- D \tag{116}$$

Using this with lemma 19 yields

$$t <:^- g \quad \text{or} \quad t = D$$

for some ground type g.

Case $t = D$ Derive the subtype relation by

$$\frac{}{D <: D}$$

Case $t \neq D \wedge t <:^- g$ Using lemma 21 yields

$$t <: g' \tag{117}$$

Derive the subtype relation by

$$\frac{(117)}{t <: D}$$

□

Case $t_1 \otimes t_2 <:^+ t'_1 \otimes t'_2$ The assumptions are

$$t_1 \otimes t_2 <:^- t'_1 \otimes t'_2 \tag{118}$$
$$t_1 <:^+ t'_1 \tag{119}$$
$$t_2 <:^+ t'_2 \tag{120}$$

Assumption (118) could only be derived by two rules:

$$\frac{t_1 <:^- t_1' \quad t_2 <:^+ t_2'}{t_1 \otimes t_2 <:^- t_1' \otimes t_2'} \tag{121}$$

or

$$\frac{t_1 \otimes t_2 <:^- g}{t_1 \otimes t_2 <:^- t_1' \otimes t_2'} \tag{122}$$

for some ground-type g.

Case (121) We have $t_1 <:^- t_1'$ and $t_2 <:^- t_2'$. Using this with (119) and (120) and the induction assumption yields

$$t_1 <: t_1' \tag{123}$$
$$t_2 <: t_2' \tag{124}$$

Derive the subtype relation by:

$$\frac{(123) \quad (124)}{t_1 \otimes t_2 <: t_1' \otimes t_2'}$$

Case (122) We have $t_1 \otimes t_2 <:^- g$ for some ground-type g. Using this with lemma 20 yields

$$t_1 <:^- g' \quad \text{or} \quad t_1 = D \tag{125}$$
$$t_2 <:^- g'' \quad \text{or} \quad t_2 = D \tag{126}$$

for some ground types g', g''. From (125) it follows that

$$t_1 <:^- t_1' \tag{127}$$

by deriving either

$$\frac{t_1 <:^- g'}{t_1 <:^- t_1'}$$

or

$$\frac{}{D <:^- t_1'}$$

Similarly it follows from (126) that

$$t_2 <:^- t_2' \tag{128}$$

Using (127), (128), (119), (120) and the induction assumptions yields

$$t_1 <: t_1' \tag{129}$$
$$t_2 <: t_2' \tag{130}$$

and the subtype relation can be derived by:

$$\frac{(129) \quad (130)}{t_1 \otimes t_2 <: t_1' \otimes t_2'}$$

□

The case $!t <:^+ !t'$ works similarly.

Case $t_1 \multimap t_2 <:^+ t_1' \multimap t_2'$ The assumptions are

$$t_1 \multimap t_2 <:^- t_1' \multimap t_2' \tag{131}$$
$$t_1' <:^- t_1 \tag{132}$$
$$t_2 <:^+ t_2' \tag{133}$$

Assumption (131) could only be derived by two rules:

$$\frac{t_1' <:^+ t_1 \quad t_2 <:^- t_2'}{t_1 \multimap t_2 <:^- t_1' \multimap t_2'} \tag{134}$$

or

$$\frac{t_1 \multimap t_2 <:^- g}{t_1 \multimap t_2 <:^- t_1' \multimap t_2'} \tag{135}$$

for some ground-type g.

Case (134) We have $t_1' <:^+ t_1$ and $t_2 <:^- t_2'$. Using this with (132) and (133) and the induction assumption yields

$$t_1' <: t_1 \tag{136}$$
$$t_2 <: t_2' \tag{137}$$

Derive the subtype relation by:

$$\frac{(136) \quad (137)}{t_1 \multimap t_2 <: t_1' \multimap t_2'}$$

Case (135) We have $t_1 \multimap t_2 <:^- g$ for some ground-type g. Using this with lemmas 20 and 19 yields

$$t_1 = D \tag{138}$$
$$t_2 <:^- g' \quad \text{or} \quad t_2 = D \tag{139}$$

for some ground types g'. From (139) it follows that

$$t_2 <:^- t_2' \tag{140}$$

by deriving either

$$\frac{t_2 <:^- g'}{t_2 <:^- t_2'}$$

or

$$\frac{}{D <:^{-} t_2'}$$

It follows from (138) that

$$t_1' <:^{+} t_1 \tag{141}$$

by deriving

$$\frac{}{t_1' <:^{+} D}$$

Using (140), (141), (132), (133) and the induction assumptions yields

$$t_1' <: t_1 \tag{142}$$
$$t_2 <: t_2' \tag{143}$$

and the subtype relation can be derived by:

$$\frac{(142) \quad (143)}{t_1 \multimap t_2 <: t_1' \multimap t_2'}$$

I.2 If $t <: t'$ then $t <:^{+} t'$ and $t <:^{-} t'$

This direction is proven by an induction on the derivation of $t <: t'$

Case $\star <: \star$ Derive negative and positive subtypes directly:

$$\frac{}{\star <:^{-} \star}$$
$$\frac{}{\star <:^{+} \star}$$

Case $D <: D$ Derive negative and positive subtypes directly:

$$\frac{}{D <:^{-} D}$$
$$\frac{}{D <:^{+} D}$$

Case $t <: D$ when $t <: g$ The assumptions are

$$t <: g \tag{144}$$

for some ground-type g. Using the induction assumptions with (144) yields

$$t <:^{-} g \tag{145}$$

which enables the derivation of the negative subtype relation:

$$\frac{(145)}{t <:^{-} D}$$

The positive subtype relation can be derived directly:

$$\frac{}{t <:^{+} D}$$

□

Case $t_1 \otimes t_2 <: t'_1 \otimes t'_2$ From the assumptions

$$t_1 <: t'_1$$
$$t_2 <: t'_2$$

it follows by induction assumptions that

$$t_1 <:^+ t'_1 \tag{146}$$
$$t_1 <:^- t'_1 \tag{147}$$
$$t_2 <:^+ t'_2 \tag{148}$$
$$t_2 <:^- t'_2 \tag{149}$$

The negative subtype relation can be derived:

$$\frac{(147) \quad (149)}{t_1 \otimes t_2 <:^- t'_1 \otimes t'_2}$$

The positive subtype relation can be derived:

$$\frac{(146) \quad (148)}{t_1 \otimes t_2 <:^+ t'_1 \otimes t'_2}$$

□

The cases $t_1 \multimap t_2 <: t'_1 \multimap t'_2$ and $!t <: !t'$ work similarly.

J Proof of Lemma 20

The proof is an induction on the derivation of $t <:^- g$, where g is a ground type. It is done here for the first statement of the lemma, $t_1 \multimap t_2 <:^- g$, the other proofs work similarly.

The rules to consider in this case are

$$\frac{}{\star <:^- \star}\ (1) \quad \frac{}{D <:^- g}\ (2) \quad \frac{t <:^- g}{t <:^- t'}\ (3)$$

$$\frac{t_1 <:^- D \quad t_2 <:^- D}{t_1 \otimes t_2 <:^- D \otimes D}\ (4) \quad \frac{D <:^+ t_1 \quad t_2 <:^- D}{t_1 \multimap t_2 <:^- D \multimap D}\ (5) \quad \frac{t <:^- D}{!t <:^- !D}\ (6)$$

Rules (1), (2), (4) and (6) do not fit the assumption $t_1 \multimap t_2 <:^- g$ syntactically.

J.1 Case (3)

The assumptions in this case are

$$t_1 \multimap t_2 <:^- g'$$

for some ground-type g'. Applying the induction assumption yields the desired result to satisfy the lemma.

J.2 Case (4)

As $D \multimap D$ is the only suitable ground-type in this case, the assumptions in this case are

$$D <:^+ t_1 \tag{150}$$

$$t_2 <:^- D \tag{151}$$

Lemma 4 with (150) yields $t_1 = D$. The result (151) could only be derived by rule (2) or rule (3). In case of rule (2) we have $t_2 = D$ immediately. In case of rule (3) we have $t_2 <:^- g$ for some ground-type g as the requirement.

□

K Proof of Lemma 21

The proof is an induction on $t <:^- g$ for a ground type g. The rules to consider in this case are

$$\frac{}{\star <:^- \star}\ (1) \quad \frac{}{D <:^- g}\ (2) \quad \frac{t <:^- g'}{t <:^- g}\ (3)$$

$$\frac{t_1 <:^- D \quad t_2 <:^- D}{t_1 \otimes t_2 <:^- D \otimes D}\ (4) \quad \frac{D <:^+ t_1 \quad t_2 <:^- D}{t_1 \multimap t_2 <:^- D \multimap D}\ (5) \quad \frac{t <:^- D}{!t <:^- !D}\ (6)$$

where g' is a ground-type.

K.1 Case (1)

Derive subtype relation immediately:

$$\frac{}{\star <: \star}$$

□

K.2 Case (2)

$t = D$ immediately satisfied.

K.3 Case (3)

The assumption in this case is $t <:^- g'$. Applying the induction assumption immediately yields the desired result.

K.4 Case (5)

The assumptions are

$$D <:^+ t_1 \tag{152}$$

$$t_2 <:^- D \tag{153}$$

Using lemma 4 with (152) yields

$$t_1 = D$$

from which it follows

$$t_1 <: D \tag{154}$$

by deriving

$$\frac{}{D <: D}$$

Using lemma 19 with (153) yields

$$t_2 = D \quad \text{or} \quad t_2 <:^{-} g''$$

for some ground-type g''. From this result it follows that

$$t_2 <: D \tag{155}$$

by either deriving

$$\frac{}{D <: D}$$

when $t_2 = D$ or by using the induction assumption when $t_2 <:^{-} g''$. Now the subtype relation can be derived:

$$\frac{(154) \quad (155)}{t_1 \multimap t_2 <: D \multimap D}$$

□

The cases (4) and (6) work similarly.

L Proof Extensions for Shortcut Casts

In the following section we give the additional proof cases that arise when adding the reduction rules for shortcut casts, the modified compatibility and the added subtyping rules.

L.1 Additional Lemmas

Lemma 12 (Invertible typing rules) The last case about the inversion of the *Cast* rule has to be modified to:

$$\text{If } \Theta; \Gamma \vdash \langle t \Leftarrow t' \rangle^p e : t \text{ then } \Theta; \Gamma \vdash e : t' \text{ and } t' \lesssim t$$

The proof remains straightforward.

Lemma 13 (Invertible compatibility) Type compatibility is still syntax directed. All rules are invertible.

Lemma 14 (Typing preservation under substitution) Also the substitution lemma is straightforwardly preserved with the modified compatibility.

Lemma 18 (Positive subtyping is unique) The applicability of the added rules for positive subtyping does not overlap with the standard rules. The cases are extended to

4. If $!t <:^+ \star$ then $t <:^+ \star$
5. $!t <:^+ t_1' \multimap t_2'$ then $t <:^+ t_1' \multimap t_2'$
6. $!t <:^+ t_1' \otimes t_2'$ then $t <:^+ t_1' \otimes t_2'$

Lemma 20 (Constructed negative subtypes) The result still holds. The additional cases for $!t <:^- g$ are

$$\frac{t <:^- \star}{!t <:^- \star} \qquad \frac{t <:^- t_1 \multimap t_2}{!t <:^- t_1 \multimap t_2} \qquad \frac{t <:^- t_1 \otimes t_2}{!t <:^- t_1 \otimes t_2}$$

In all cases it follows that $t <:^- g$ and therefore $t <:^- D$ can be derived.

In presence of the extended rules an additional result can be shown for the case $!t <:^- g$.

Lemma 23. *If* $!t <:^- g$ *then* $t <:^- g$ *or* $t = D$.

Proof. By induction on the derivation of $!t <:^- g$. In case

$$\frac{t <:^- D}{!t <:^- !D}$$

it follows by lemma 19 that $t = D$ or $t <:^- g'$ for a ground type g'. In the latter case, $t <:^- !D$ can be derived.

In case

$$\frac{!t <:^- g'}{!t <:^- g}$$

it follows by the induction assumptions that $t = D$ or $t <:^- g'$. In the latter case $t <:^- g$ can be derived.

In the other cases, the result follows directly.

Lemma 21 In all additional cases we can follow from $!t <:^- g$ for a ground type g that $t <:^- g$ by inverting the corresponding negative subyting rule. It then follows by induction assumption that $t <: g$. The results $!t <: g$ then follows from the corresponding additional rule for subtyping.

L.2 Type Preservation

Cases *Cast - $\multimap$*, *Cast - $\otimes$*, *Cast - !* These cases work analogously, also with the modified compatibility relation, as compatibility is contravariant in the $\multimap$ case and covariant in the ! and $\otimes$ cases. This is exactly as required by the typing derivation that concludes each of the cases.

Case *FromDyn - !t* The considered rule is

$$\{H, a = D_!(a'), a' =!x\}\langle t \Leftarrow D\rangle^p a \quad \rightarrow \{H\}\langle t \Leftarrow D\rangle^p x \quad \text{if } t \neq !t'$$

The assumptions are

$$\vdash H, a = D_!(a'), a' =!x : \Theta; a : D, \Delta \tag{156}$$
$$\Theta; a : D \vdash \langle t \Leftarrow D\rangle^p a : t \tag{157}$$

It follows from assumption (156) and lemma 8 that there exists Δ' such that

$$\vdash H, a' =!x : \Theta; \Delta', \Delta \tag{158}$$
$$\Theta; \Delta' \vdash D_!(a') : t' \tag{159}$$

It follows from (159) that $\Delta' = a' :!D$ and $t' = D$. Now, using result (156) with lemma 8 yields that there exists Δ'' such that

$$\vdash H : \Theta; \Delta'', \Delta \tag{160}$$
$$\Theta; \Delta'' \vdash !x : t'' \tag{161}$$

It follows from (161) that $\Delta'' =$ - and $t =!D$ and $\Theta = \Theta', x : D$ and

$$\Theta', x : D; \text{ - } \vdash x : D \tag{162}$$

. Heap typing is therefore given by (161) and expression typing can be derived:

$$\frac{(162) \quad t \lesssim D}{\Theta, \Theta', x : D; \text{ - } \vdash \langle t \Leftarrow D\rangle^p x : t}$$

Cases *Cast* - ! - $\star$, *Cast* - ! - $\multimap$, and *Cast* - ! - $\otimes$ These cases work similarly as the regular $! - E$ - case. The cast is well-typed by the extended compatibility relation.

L.3 Typing Progress

The only cases affected by the extensions are the cast expressions $\langle t' \Leftarrow !t \rangle^p a$, where t' is either $t'_1 \multimap t'_2$, $t'_1 \otimes t'_2$ or $\star$, and $\langle !t \Leftarrow D \rangle^p a$. In the former case it follows by lemma 10 that the heap can provide a $!x$ binding for a, and therefore one of the *Case* - $! - c$ rules can be applied for reduction. In the latter case it follows by 10 that the heap provides some dynamic value $s = D_c(a')$. In case $c \neq !$ the respective *Cast* - *Fail* - c rule can be applied. In case $c = !$ it follows by 10 that there exists $s' = !x$. Thus rule *From* - *Dyn* - ! can be applied.

L.4 Preservation of Blame Safety

In case of the *From* - *Dyn* - ! the cast is preserved by the reduction and its arguments are variables, thus the result is safe. In case of the *Case* - ! - c rules, the respective subtyping relations are preserved in the casts as subtyping was appropriately extended.

L.5 Subtype Factoring

The previously established cases remain valid as the additional lemmas about subtyping all still hold. In the case

$$\frac{t <:^+ t'_1 \multimap t'_2}{!t <:^+ t'_1 \multimap t'_2}$$

we have $!t <:^- t'_1 \multimap t'_2$ or $!t <:^- g$ for some ground type g. The former case the induction assumption directly yields $t <: t'_1 \multimap t'_2$ which allows to derive $!t <: t'_1 \multimap t'_2$. In the latter case it follows by lemma 23 that $t = D$ or $t <:^- g$. The case $t = D$ can be excluded by the assumptions and lemma 4. It follows from $t <:^- g$ that $t <:^- t'_1 \multimap t'_2$ which enable the induction assumption. The other cases work similarly.

The opposite direction of factoring also follows straightforwardly.