Analysis and Caching of Dependencies

Martín Abadi
Digital Systems Research Center
ma@pa.dec.com

Butler Lampson
Microsoft
blampson@microsoft.com

Jean-Jacques Lévy
INRIA Rocquencourt
Jean-Jacques.Levy@inria.fr

Abstract

We address the problem of dependency analysis and caching in the context of the λ-calculus. The dependencies of a λ-term are (roughly) the parts of the λ-term that contribute to the result of evaluating it. We introduce a mechanism for keeping track of dependencies, and discuss how to use these dependencies in caching.

1 Introduction

Suppose that we have evaluated the function application $f(1,2)$, and that its result is 7. If we cache the equality $f(1,2) = 7$, we may save ourselves the work of evaluating $f(1,2)$ in the future. Suppose further that, in the course of evaluating $f(1,2)$, we noticed that the first argument of f was not accessed at all. Then we can make a more general cache entry: $f(n,2) = 7$ for all n. In call-by-name evaluation, we may not even care about whether n is defined or not. Later, if asked about the result of $f(2,2)$, for example, we may match $f(2,2)$ against our cache entry, and deduce that $f(2,2) = 7$ without having to compute f.

There are three parts in this caching scheme: (i) the dependency analysis (in this case, noticing that f did not use its first argument in the course of the computation); (ii) writing down dependency information, in some way, and caching it; (iii) the cache lookup. Each of the parts can be complex. However, the caching scheme is worthwhile if the computation of f is expensive and if we expect to encounter several similar inputs (e.g., $f(1,2)$, $f(2,2)$, ...).

We address the problem of dependency analysis and caching in the context of the λ-calculus. We introduce a mechanism for keeping track of dependencies, and show how to use these dependencies in caching. (However, we stop short of considering issues of cache organization, replacement policy, etc.) Our techniques apply to programs with higher-order functions, and not just to trivial first-order examples like $f(1,2)$. The presence of higher-order functions creates the need for sophisticated dependency propagation.

As an example, consider the higher-order function:

$$f \quad \triangleq \quad \lambda x.\lambda y.fst(x(fst(y))(snd(y)))$$

where pairs are encoded as usual:

$$\begin{aligned} \langle a, b\rangle &\triangleq \lambda x.x(a)(b) \\ fst &\triangleq \lambda p.p(\lambda u.\lambda z.u) \\ snd &\triangleq \lambda p.p(\lambda u.\lambda z.z) \end{aligned}$$

The function f takes two arguments x and y; presumably x is a function and y is a pair. The function f applies x to the first and second components of y, and then extracts the first component of the result. A priori, it may seem that f depends on x and on all of y. Consider now the following arguments for f:

$$\begin{aligned} g &\triangleq \lambda u.\lambda z.\langle z, u\rangle & r &\triangleq \langle 1, 2\rangle \\ g' &\triangleq \lambda u.\lambda z.\langle z, \langle u, z\rangle\rangle & r' &\triangleq \langle 2, 2\rangle \end{aligned}$$

Both functions g and g' seem to depend on their respective arguments. However, all these a priori expectations are too coarse. After evaluating $f(g)(r)$ to 2, we can deduce that $f(g')(r')$ also yields 2. For this we need to express that f accesses only part of the pair that g produces, that g accesses only part of the pair that f feeds it, and that g and g' look sufficiently similar. We develop a simple way of capturing and of exploiting these fairly elaborate dependencies.

Our approach is based on a labelled λ-calculus [Lév78]. Roughly, our labelled λ-calculus is like a λ-calculus with names for subexpressions. In the course of computation, the names propagate, and some of them end up in the result. If a reduces to v, then v will contain the names of the subexpressions of a that contribute to producing v. Then, if we are given a' that coincides with a on those subexpressions, we may deduce that a' reduces to v.

In our example, we would proceed as follows. First, when given the expression $f(g)(r)$, we would label some of its subexpressions. The more labels we use, the more information we obtain. In this example, which is still relatively simple, we label only components of g and r:

$$\hat{g} \triangleq \lambda u.\lambda z.\langle e_0{:}z, e_1{:}u\rangle \qquad \hat{r} \triangleq \langle e_2{:}1, e_3{:}2\rangle$$

where e_0, e_1, e_2, and e_3 are distinct labels. We extend the reduction rules of the λ-calculus to handle labels; in this case, $f(\hat{g})(\hat{r})$ reduces to $e_0{:}e_3{:}2$. Stripping off all the labels, we can deduce that $f(g)(r)$ reduces to 2. Studying the labels, we may notice that e_1 and e_2 do not appear in the result. As we will prove, this means that $f(g^*)(r^*)$ reduces to 2 for any expressions g^* and r^* of the forms:

$$g^* \triangleq \lambda u.\lambda z.\langle z, \_\rangle \qquad r^* \triangleq \langle \_, 2\rangle$$

Obviously, g' and r' match this pattern, and hence $f(g')(r')$ reduces to 2. As this small example suggests, our techniques for dependency analysis are effective, reasonably efficient, and hence potentially practical.

In the next section we review the background for our work and some related work. In section 3, we study dependency analysis and caching in the pure λ-calculus. In sections 4, we extend our techniques to a more realistic language; this language includes records and has a weak operational semantics based on explicit substitutions [ACCL91, Fie90].

2 Motivation and Related Work

The motivation for this work arose in the context of a system-modelling system called Vesta [LM93, HL93]—roughly a replacement for tools like make and rcs. In Vesta, the analogue of a makefile is a program written in a specialized, untyped, higher-order, functional, lazy, interpreted language. The functional character of the language guarantees that the results of system building are predictable and reproducible.

In Vesta, the basic computation steps are expensive calls to functions like compile and link; hence it is important to avoid unnecessary recomputations. The programs can be reasonably large; it is therefore desirable to notice cache hits for large subexpressions rather than for individual calls to primitives (e.g., individual compilations). Furthermore, irrelevant changes in parameters are expected to be frequent; when there are such changes, a simple memoisation [Mic68, Hug85] does not suffice for avoiding recomputations, and a more savvy caching strategy is necessary.

This paper, however, is not about Vesta. There has been some research on caching in Vesta [HL93], and more is currently in progress. Here we discuss techniques for the λ-calculus; these are somewhat simpler, easier to explain, and perhaps of more general interest.

In the λ-calculus, the work that seems most closely related to ours is that of Field and Teitelbaum [FT90]. They have investigated the problem of reductions of similar expressions (which may not even yield the same result). Their approach is based on a λ-calculus with a new "fork" primitive (Δ) rather than on a labelled λ-calculus. For example, they can represent the two similar expressions $b(a)$ and $b'(a)$ as the single expression $\Delta(b,b')(a)$, with a rule for duplication, namely $\Delta(b,b')(a) = \Delta(b(a),b'(a))$. Their algorithm seems particularly appropriate for dealing with pairs of expressions that differ at only one or a few predictable subexpressions.

There has been much other work on incremental computation, and some of it is related to ours. In particular, Pugh's dissertation concerns incremental evaluation of functional programs; it raises the issue of caching for functions that do not depend on all of their arguments [Pug88, pp. 70–71].

Dependency analysis is also similar to traditional analyses such as strictness analysis (e.g., [BHA86]). There is even a recent version of strictness analysis that relies on a labelled λ-calculus [GVS95]. Strictness analysis is concerned with what parts of a program *must* be evaluated; in contrast, for doing cache lookups, we need to know what parts of a program *may* affect the result. Furthermore, we do not use approximate abstract interpretations, but rather rely on previous, actual executions of programs similar to the one being analyzed.

3 Dependencies in the Pure λ-calculus

In this section we consider incremental computation in the context of the pure λ-calculus. This is a minimal system, but it enables us to illustrate our ideas. First we review some classical results that suggest an approach to dependency analysis; then we describe a labelled calculus, a basic scheme for caching, some examples, and a more sophisticated scheme for caching; finally, we consider call-by-value evaluation.

3.1 The λ-calculus

The standard λ-calculus has the following grammar for expressions:

$$\begin{array}{lcll} a,b,c & ::== & & \text{terms} \\ & \mid & x & \text{variable } (x \in V) \\ & \mid & \lambda x.a & \text{abstraction } (x \in V) \\ & \mid & b(a) & \text{application} \end{array}$$

where V is a set of variables.

The β rule is, as usual:

$$(\lambda x.b)a \quad \rightarrow \quad b\{a/x\}$$

where $b\{a/x\}$ is the result of replacing x with a in b. When C is a context (a term with a hole), we write $C\{a\}$ for the result of filling C's hole with a (possibly with variable captures). We adopt the following congruence rule:

$$\frac{a \rightarrow b}{C\{a\} \rightarrow C\{b\}}$$

The reduction relation $\rightarrow^\star$ is the reflexive, transitive closure of $\rightarrow$. A computation stops when it reaches a normal form.

We can now reformulate the problem posed in the introduction. Suppose that a is a term and $a \rightarrow^\star v$. When can we say that $b \rightarrow^\star v$ simply by comparing a and b? In order to address this question, we recall a few known theorems.

Theorem 1 (Church-Rosser) *The calculus is confluent.*

Theorem 2 (Normalization) *If a term can be reduced to a normal form, then its leftmost outermost reduction (which reduces the leftmost outermost redex at each step) reaches this normal form.*

Clearly the leftmost outermost reduction reduces only subexpressions necessary to get to the normal form.

A prefix is an expression possibly with several missing subexpressions:

$$\begin{array}{lcll} a,b,c & ::== & & \text{prefixes} \\ & \mid & \_ & \text{hole} \\ & \mid & x & \text{variable } (x \in V) \\ & \mid & \lambda x.a & \text{abstraction } (x \in V) \\ & \mid & b(a) & \text{application} \end{array}$$

A prefix a is prefix of another prefix (or expression) b if a matches b except in some holes; we write $a \preceq b$. For instance, we have that $\_(x)(\_)(\lambda y.\_(y)) \preceq y(x)(x)(\lambda y.\_(y))$. For the purposes of reduction, we treat $\_$ like a free variable; for example, $(\lambda x.x(\_))(a) \rightarrow a(\_)$.

The following three results concern the prefix ordering and reduction:

Proposition 1 (Maximality of terms) *If $b \preceq d$ and b is a term, then $b = d$.*

Proposition 2 (Monotonicity) *If a, b, and c are prefixes, $a \rightarrow^\star b$, and $a \preceq c$, then there exists a prefix d such that $c \rightarrow^\star d$ and $b \preceq d$.*

Theorem 3 (Stability) *If a is a term, v is a term in normal form, and $a \rightarrow^\star v$, then there is a minimum prefix $a_0 \preceq a$ such that $a_0 \rightarrow^\star v$.*

Proof The stability theorem follows from the stability of Böhm trees [Ber78]. Here we sketch a simple, alternative proof.

First we show that if a and b are compatible (have a common upper bound) in the prefix ordering, and a and b reduce to a term v in normal form, then the greatest lower bound of a and b (written $a \wedge b$) also reduces to v. The proof is by induction on the lengths of the leftmost outermost reductions to v, and secondarily on the sizes of a and b. We proceed by cases on the form of a.

- If $a = x$, then $v = x$ and $b = x$, so $(a \wedge b) = v$.
- If $a = \lambda x.a_1$, then b is of the form $\lambda x.b_1$, with a_1 and b_1 compatible. The result follows by induction hypothesis.
- If $a = x(a_1)\dots(a_n)$, then b is of the form $x(b_1)\dots(b_n)$ with a_i and b_i compatible for each $i \in 1..n$. The result follows by induction hypothesis.
- The cases where $a = \_$ or $a = \_(a_1)\dots(a_n)$ are impossible, since a reduces to a term in normal form.
- Finally, if $a = (\lambda x.a_1)(a_2)\dots(a_n)$, then b is of the form $(\lambda x.b_1)(b_2)\dots(b_n)$. Let $a' = a_1\{a_2/x\}(a_3)\dots(a_n)$ and $b' = b_1\{b_2/x\}(b_3)\dots(b_n)$; a' and b' are compatible, and they reduce to v with shorter leftmost outermost reductions than a and b. By induction hypothesis, $a' \wedge b'$ reduces to v. Since $a \wedge b$ reduces to $a' \wedge b'$, we obtain that $a \wedge b$ reduces to v by transitivity.

Now suppose that a and v are as indicated in the statement of the theorem. The prefixes of a that reduce to v are compatible, since they have a as upper bound; their greatest lower bound is the prefix a_0 described in the statement. □

We can give a first solution to our problem, as follows. Suppose that $a \rightarrow^\star v$ and v is a term in normal form. Let a_0 be the minimum prefix of a such that $a_0 \rightarrow^\star v$, given by Theorem 3. By Proposition 2, if a_0 is a prefix of b then $b \rightarrow^\star v'$ for some v' such that v is a prefix of v'; by Proposition 1, v' is v. Therefore, if a_0 is a prefix of b then we can reuse the computation $a \rightarrow^\star v$ and conclude that $b \rightarrow^\star v$.

It remains for us to compute a_0. As we will show, this computation can be performed at the same time as we evaluate a, and does not require much additional work. Intuitively, we will mark every subexpression of a necessary to compute v along the leftmost outermost reduction.

3.2 A labelled λ-calculus

In order to compute minimum prefixes as discussed above, we follow the underlined method of Barendregt [Bar84], generalized by use of labels as in the work of Field, Lévy, or Maranget [Fie90, Lév78, Mar91]. Our application of this method gives rise to a new labelled calculus, which we define next.

We consider a λ-calculus with the following extended grammar for expressions:

$$\begin{array}{lcll} a, b, c & ::== & & \text{terms} \\ & | & \dots & \text{as in section 3.1} \\ & | & e{:}a & \text{labelled term } (e \in E) \end{array}$$

where E is a set of labels.

There is one new one-step reduction rule:

$$(e{:}b)(a) \quad \rightarrow \quad e{:}(b(a))$$

The essential purpose of this rule is to move labels outwards as little as possible in order to permit β reduction. For example, $(e_0{:}(\lambda x.x(x)))(e_1{:}y)$ reduces to $e_0{:}((\lambda x.x(x))(e_1{:}y))$ via the new rule, and then yields $e_0{:}((e_1{:}y)(e_1{:}y))$ by the β rule.

There are clear correspondences between the unlabelled calculus and the labelled calculus. When a' is a labelled term, let *strip(a')* be the unlabelled term obtained by removing every label in a'. We have:

Proposition 3 (Simulation) *Let a, b be terms, and let a', b' be labelled terms.*

- *If $a' \rightarrow b'$, then strip(a') $\rightarrow^\star$ strip(b').*
- *If a = strip(a') and $a \rightarrow b$, then $a' \rightarrow^\star b'$ for some b' such that b = strip(b').*

The labelled calculus enjoys the same fundamental theorems as the unlabelled calculus: confluence, normalization, and stability. The confluence theorem follows from Klop's dissertation work, because the labelled calculus is a regular combinatory reduction systems [Klo80]; the labelled calculus is left-linear and without critical pairs. The normalization theorem can also be derived from Klop's work; alternatively it can be obtained from results about abstract reductions systems [GLM92], via O'Donnell's notion of left systems [O'D77]. The proof of the stability theorem is similar to the one in [HL91].

3.3 Basic caching

Suppose that $a \rightarrow^\star v$, where a is a term and v is its normal form. Put a different label on every subexpression of a, obtaining a labelled term a'. By Proposition 3, $a' \rightarrow^\star v'$ for some v' such that v = *strip(v')*. Consider all the labels in v'; to each of these labels corresponds a subterm of a' and thus of a. Let $G(a)$ be a prefix obtained from a by replacing with $\_$ each subterm whose label does not appear in v'. We can prove that $G(a)$ is well-defined. In particular, the value of $G(a)$ does not depend on the choice of a' or v'; and if the label for a subterm of a appears in v' then so do the labels for all subterms that contain it.

When $a \rightarrow^\star v$, we may cache the pair $(G(a), v)$. When we consider a new term b, it is sufficient to check that $G(a) \preceq b$ in order to produce v as the result of b. As $G(a)$ is the part of a sufficient to get v (what we called a_0 in section 3.1), we obtain the following theorem:

Theorem 4 *If a is a term, v is a term in normal form, $a \rightarrow^\star v$, and $G(a) \preceq b$, then $b \rightarrow^\star v$.*

Theorem 4 supports a simple caching strategy. In this strategy, we maintain a cache with the following invariants:

- the cache is a set of pairs (a_0, v), consisting each of an unlabelled prefix a_0 and an unlabelled term v in normal form;

- if (a_0, v) is in the cache and $a_0 \preceq b$ then $b \rightarrow^{\star} v$.

Therefore, whenever we know that v is the normal form of a, we may add to the cache the pair $(G(a), v)$. Theorem 4 implies that this preserves the cache invariants.

Suppose that a is a term without labels. In order to evaluate a, we do:

- if there is a cache entry (a_0, v) such that $a_0 \preceq a$, then return v;

- otherwise:
  - let a' be the result of adding distinct labels to a, at every subexpression;
  - suppose that, by reduction, we find that $a' \rightarrow^{\star} v'$ for v' in normal form;
  - let $v = \mathit{strip}(v')$ and $a_0 = G(a)$;
  - optionally, add the entry (a_0, v) to the cache;
  - return v.

Both cases preserve the cache invariants. In both, the v returned is such that $a \rightarrow^{\star} v$.

In a refinement of this scheme, we may put labels at only some subexpressions of a. In this case, we replace with _ a subexpression of a only if this subexpression was labelled before reduction. The more labels we use, the more general the prefix obtained; this results in better cache entries, at a moderate cost. However, in examples, we prefer to use few labels in order to enhance readability.

Another refinement of the scheme consists in caching pairs of labelled prefixes and results. The advantage of not stripping the labels is that the cache records the precise dependencies of results on prefixes. We return to this subject in section 3.5.

3.4 Examples

The machinery that we have developed so far handles the example of the introduction (the term $f(g)(r)$). We leave the step-by-step calculation for that example as an exercise to the reader. As that example illustrates, pairing behaves nicely, in the sense that $\mathit{fst}\langle a, b\rangle$ depends only on a, as one would expect.

As a second example, we show that the Church booleans behave nicely too. The encoding of booleans is as usual:

$$\begin{aligned} \text{true} &\triangleq \lambda x.\lambda y.x \\ \text{false} &\triangleq \lambda x.\lambda y.y \\ \text{if } a \text{ then } b \text{ else } c &\triangleq a(b)(c) \end{aligned}$$

In the setting of the labelled λ-calculus, we obtain as a derived rule that:

$$\text{if } (e{:}a) \text{ then } b \text{ else } c \rightarrow^{\star} e{:}(\text{if } a \text{ then } b \text{ else } c)$$

It follows from this rule that, for example,

$$(\lambda x.\, \text{if } e_0{:}x \text{ then } e_1{:}y \text{ else } e_2{:}z)(e_3{:}\text{true}) \rightarrow^{\star} e_0{:}e_3{:}e_1{:}y$$

We obtain the unlabelled prefix:

$$(\lambda x.\, \text{if } x \text{ then } y \text{ else } \_)(\text{true})$$

and we can deduce that any expression that matches this prefix reduces to y.

Similar examples arise in the context of Vesta (see section 2 and [HL93]). A simple one is the term:

$$(\text{if } \mathit{isC}(\mathit{file}) \text{ then } \mathit{Ccompile} \text{ else } \mathit{M3compile})(\mathit{file})$$

where $\mathit{isC}(f)$ returns true whenever f is a C source file, and *file* is either a C source file or an M3 source file. If $\mathit{isC}(\mathit{file})$ returns true, then that term yields $\mathit{Ccompile}(\mathit{file})$. Using labels, we can easily discover that this result does not depend on the value of *M3compile*, and hence that it need not be recomputed when that value changes. In fact, even $\mathit{isC}(\mathit{file})$ and the conditional need not be reevaluated.

In a higher-order variant of this example, the conditional compilation function is passed as an argument:

$$\begin{aligned} &(\lambda x.\, x(\mathit{file})) \\ &\quad (\lambda y.\, \text{if } \mathit{isC}(y) \text{ then } \mathit{Ccompile}(y) \text{ else } \mathit{M3compile}(y)) \end{aligned}$$

Our analysis is not disturbed by the higher-order abstraction, and yields the same information.

3.5 Limitations of the basic caching scheme

The basic caching scheme of section 3.3 has some limitations, illustrated by the following two concrete examples.

Suppose that we have the cache entry:

$$((\lambda x.\langle \mathit{snd}(x), \mathit{fst}(x)\rangle)(\langle \text{true}, \text{false}\rangle), \langle \text{false}, \text{true}\rangle)$$

Suppose further that we wish to evaluate the term:

$$\mathit{fst}((\lambda x.\langle \mathit{snd}(x), \mathit{fst}(x)\rangle)(\langle \text{true}, \text{false}\rangle))$$

Immediately the cache entry enables us to reduce this term to $\mathit{fst}(\langle \text{false}, \text{true}\rangle))$, and eventually we obtain false. However, in the course of this computation, we have not learned how the result depends on the input. We are unable to make an interesting cache entry for the term we have evaluated. Given the new, similar term

$$\mathit{fst}((\lambda x.\langle \mathit{snd}(x), \mathit{fst}(x)\rangle)(\langle \text{false}, \text{false}\rangle))$$

we cannot immediately tell that it yields the same result.

As a second example, suppose that we have the cache entry:

$$(\text{if true then true else } \_, \text{true})$$

and that we wish to evaluate the term:

$$\text{not}(\text{if true then true else true})$$

In our basic caching scheme, we would initially label this term, for example as:

$$\text{not}(\text{if true then } e_0{:}\text{true else } e_1{:}\text{true})$$

Then we would have to reduce this term, and as part of that task we would have to reduce the subterm

$$(\text{if true then } e_0{:}\text{true else } e_1{:}\text{true})$$

At this point our cache entry would tell us that the subterm yields true, modulo some labels. We can complete the reduction, obtaining false, and we can make a trivial cache entry:

$$(\text{not}(\text{if true then true else true}), \text{false})$$

However, we have lost track of which prefix of the input determines the result, and we cannot make the better cache entry:

$$(\text{not}(\text{if true then true else }\_), \text{false})$$

The moral from these examples is that cache entries should contain dependency information that indicates how each part of the result depends on each part of the input. One obvious possibility is not to strip the labels of prefixes and results before making cache entries; after all, these labels encode the desired dependency information. We have developed a refinement of the basic caching scheme that does precisely that, but we omit its detailed description in this paper. Next we give another solution to the limitations of the basic caching scheme.

3.6 A more sophisticated caching scheme

In this section we describe another caching scheme. This scheme does not rely directly on the labelled λ-calculus, but it can be understood or even implemented in terms of that calculus.

With each reduction $a \rightarrow^\star v$ of a term a, we associate a function d from prefixes of v to prefixes of a. Very roughly, if v_0 is a prefix of a then $d(v_0)$ is the prefix of a that yields v_0 in the reduction $a \rightarrow^\star v$. We write $a \rightarrow^\star_d v$ to indicate the function d. This annotated reduction relation is defined by the following rules.

- Reflexivity:

$$a \rightarrow^\star_{id} a$$

  where id is the identity function on prefixes.

- Transitivity:

$$\frac{a \rightarrow^\star_d b \qquad b \rightarrow^\star_{d'} c}{a \rightarrow^\star_{(d';d)} c}$$

  where $d';d$ is the function composition of d' and d.

- Congruence: Given a function d from prefixes of b to prefixes of a, we define a function $C\{d\}$ from prefixes of $C\{b\}$ to prefixes of $C\{a\}$. If $c_0 \preceq C$ then $C\{d\}(c_0) = c_0$; otherwise, there exists a unique $b_0 \preceq b$ such that $c_0 = C\{b_0\}$, and we let $C\{d\}(c_0) = C\{d(b_0)\}$. We obtain the rule:

$$\frac{a \rightarrow^\star_d b}{C\{a\} \rightarrow^\star_{C\{d\}} C\{b\}}$$

- β:

$$(\lambda x.b)(a) \rightarrow^\star_{d_\beta} b\{a/x\}$$

  where $d_\beta(\_) = \_$ and, for $c_0 \neq \_$ and $c_0 \preceq b\{a/x\}$, $d_\beta(c_0) = (\lambda x.b_0)(a_0)$ where a_0 and b_0 are the least prefixes such that $a_0 \preceq a$, $b_0 \preceq b$, and $c_0 \preceq b_0\{a_0/x\}$.

These rules are an augmentation of the reduction rules of section 3.1, in the following sense:

Proposition 4 • *If $a \rightarrow^\star_d b$ then $a \rightarrow^\star b$.*

- *If $a \rightarrow^\star b$ then $a \rightarrow^\star_d b$ for some d.*

The rules may seem a little mysterious, but they can be understood in terms of labels. Imagine that every subexpression of a is labelled (with an invisible label), that $a \rightarrow^\star_d v$, and that $v_0 \preceq v$; then $d(v_0)$ is the least prefix of a that contains all of the labels that end up in v_0.

As an example, consider the term $(\lambda x.x(x))(a)$, where a is arbitrary. By β, we have

$$(\lambda x.x(x))(a) \rightarrow^\star_{d_\beta} a(a)$$

where d_β is such that, for instance, $d_\beta(\_)$ is $\_$, $d_\beta(a(a))$ is the entire $(\lambda x.x(x))(a)$, and $d_\beta(a(\_))$ is $(\lambda x.x(\_))(a)$. If we had labelled the initial term $(\lambda x.x(x))(a)$ before reduction, then the labels that would decorate the result prefix $a(\_)$ would be all those of the initial term except for the label of the argument occurrence of x; this justifies that $d_\beta(a(\_))$ be $(\lambda x.x(\_))(a)$.

We obtain:

Theorem 5 *If a is a term, $a \rightarrow^\star_d v$, and $d(v) \preceq b$, then $b \rightarrow^\star_d v$.*

This theorem gives rise to a new caching scheme. The cache entries in this scheme consist of judgements $a \rightarrow^\star_d v$, where a and v are terms and d is a function from prefixes of v to prefixes of a. The representation of d can be its graph (i.e., a set of pairs of prefixes) or a formal expression (written in terms of id, d_β, etc.); it can even be the pair of a labelling of a and a corresponding labelling of v. According to the theorem, whenever we encounter a term b such that $d(v) \preceq b$, we may deduce that $b \rightarrow^\star_d v$.

This caching scheme does not suffer from the limitations of the basic one. In particular, each cache entry contains dependency information for every part of the result, rather than for the whole result. Moreover, the rules of inference provide a way of combining dependency information for subcomputations; therefore, we can make an interesting cache entry whenever we do an evaluation, even if we used the cache in the course of the evaluation.

3.7 Call-by-value

So far, we have considered only call-by-name evaluation. Here we define a call-by-value version of the labelled λ-calculus, showing that we can adapt our approach to call-by-value evaluation. The move from a call-by-name to a call-by-value labelled λ-calculus does not affect the basic caching scheme of section 3.3, which remains sound.

The syntax of the call-by-value labelled λ-calculus is that given in section 3.2. The β rule is restricted to:

$$(\lambda x.b)v \quad \rightarrow \quad b\{v/x\}$$

where v ranges over terms of the form x, $x(a_1)\ldots(a_n)$, or $\lambda x.a$; such terms are called values. As in section 3.2, we have a rule for moving labels outwards from the left-hand side of applications:

$$(e{:}b)(a) \quad \rightarrow \quad e{:}(b(a))$$

We have an additional rule for the right-hand side of applications:

$$(\lambda x.b)(e{:}a) \quad \rightarrow \quad e{:}((\lambda x.b)(a))$$

One might be tempted to adopt a stronger rule, namely $b(e{:}a) \rightarrow e{:}(b(a))$, but this rule creates critical pairs.

With call-by-name evaluation, the term $(\lambda x.b)(a)$ does not depend on a if x does not occur free in b. In particular, any term that has $(\lambda x.b)(\_)$ as prefix yields the same result. With call-by-value evaluation, on the other hand, the behavior of $(\lambda x.b)(a)$ depends on a: it depends on whether a can be reduced to a value or not. Therefore, a term may match the prefix $(\lambda x.b)(\_)$ but not yield the same result as $(\lambda x.b)(a)$.

The treatment of labels in our rules for call-by-value takes into account that $(\lambda x.b)(a)$ depends on a even if x does not occur free in b. Suppose that a can be reduced to a value v. To see how $(\lambda x.b)(a)$ depends on a, we add a label in front of a, obtaining $(\lambda x.b)(e{:}a)$. Labelled reduction yields $e{:}((\lambda x.b)(a))$, then $e{:}((\lambda x.b)(v))$, and finally $e{:}b$. The label e does not disappear, as it would in call-by-name evaluation.

For instance, let us consider the case where a is the value $y(z)$. From a labelled reduction we obtain the prefix $(\lambda x.b)(\_(\_))$. Any term that matches this prefix reduces to the same result as $(\lambda x.b)(y(z))$. In fact, the evaluation of $(\lambda x.b)(y(z))$ reveals that this term depends on the "valueness" of $y(z)$, but does not depend on $y(z)$ in any other way. The prefix $(\lambda x.b)(\_(\_))$ does not express this information with full accuracy, but approximates it.

4 Dependencies in a Weak λ-calculus with Records

The techniques developed in the previous section are not limited to the pure λ-calculus. In this section, we demonstrate their applicability to a more realistic language, with primitive booleans, primitive records, and explicit substitutions. The operational semantics of this language is weak (so function and record closures are not reduced).

4.1 A weak calculus with records

We consider an extended λ-calculus with the grammar for terms and for substitutions given in Figure 1. In the grammar, L is a set of names (field names for records) and *else* is a keyword used for "else" clauses in records and in substitutions. As we show below, these "else" clauses are useful in dependency analysis. We typically think of the "else" clauses as corresponding to run-time errors (missing fields, unbound variables). The term a_{n+1} in an "else" clause can be arbitrary; a term that represents a run-time error will do.

We use the following notation for extending substitutions. Let s be $x_1 = a_1, \ldots, x_n = a_n, else = a_{n+1}$; then $(x = a) \cdot s$ is $x = a, x_1 = a_1, \ldots, x_n = a_n, else = a_{n+1}$ if x is not among the variables $x_1, \ldots, x_n$, and it is $x = a, x_1 = a_1, \ldots, x_{i-1} = a_{i-1}, x_{i+1} = a_{i+1}, \ldots, x_n = a_n, else = a_{n+1}$ if x is x_i.

The one-step reduction rules now use explicit substitutions. They are given in Figure 2. In particular, the analogue for the β rule is $((\lambda x.b)[s])a \to b[(x = a) \cdot s]$, which extends an explicit substitution; the replacement of a for x happens gradually, through other rules which push the substitution inwards in b.

An active context is a context generated by the grammar of Figure 3. We adopt the following congruence rule: for any active context C,

$$\frac{a \to b}{C\{a\} \to C\{b\}}$$

Notice that this rule allows us to compute inside substitutions, but not under λ, inside records, or in the term part of closures (since $\lambda x.C$, $\langle \ldots, l_i = C, \ldots \rangle$, $\langle \ldots, else = C \rangle$, and $C[s]$ are not listed as active contexts). The relation $\to^\star$ is the reflexive, transitive closure of $\to$.

The prefix ordering for this language is interesting. Let s be the substitution $x_1 = a_1, \ldots x_n = a_n, else = a_{n+1}$, let r be the record $\langle l_1 = a_1, \ldots, l_n = a_n, else = a_{n+1} \rangle$. We associate with s and r the following functions from variables or field names to prefixes:

$$[\![s]\!](x) = \begin{cases} a_i & \text{if } x = x_i \text{ for some } i \\ a_{n+1} & \text{otherwise} \end{cases}$$

$$[\![r]\!](l) = \begin{cases} a_i & \text{if } l = l_i \text{ for some } i \\ a_{n+1} & \text{otherwise} \end{cases}$$

Intuitively, $[\![s]\!](x)$ is the image of x through s, and $[\![r]\!](l)$ is the image of l through r. The prefix ordering is as before except for substitutions and records where $s \preceq s'$ if $[\![s]\!](x) \preceq [\![s']\!](x)$ for all $x \in V$, and $r \preceq r'$ if $[\![r]\!](l) \preceq [\![r']\!](l)$ for all $l \in L$.

According to this definition, the order of the components of substitutions and records does not matter. In addition, we obtain that, if the "else" clause has a hole, then any other holes can be collapsed into it; for example, the prefixes $\langle l_1 = a, l_2 = \_, else = \_ \rangle$, $\langle l_3 = \_, l_1 = a, else = \_ \rangle$, and $\langle l_1 = a, else = \_ \rangle$ are all equivalent.

This λ-calculus enjoys the same theorems as the pure λ-calculus of section 3.1 (modulo that now $\preceq$ is actually a pre-order, not an order). These theorems should not be taken for granted, however. Their proofs are less easy, but they can be done by using results on abstract reduction systems [GLM92]. The stability theorem ensures that there is a minimum prefix for obtaining any result; moreover, the maximality and monotonicity propositions are the basis for a caching mechanism.

Finally, we should note that, in this calculus, closures may contain irrelevant bindings. For example, consider the function closure $(\lambda y.y)[x = z, else = w]$, where z is a variable and w is an arbitrary normal form. This closure reduces only to itself; the irrelevant substitution does not disappear. In this case, we will consider that the result depends on the substitution. We could add rules for minimizing substitutions but, for the sake of simplicity, we do not.

4.2 A weak labelled calculus with records

Following the same approach as in section 3, we define a labelled calculus:

$$\begin{array}{rcll} a, b, c & ::== & & \text{terms} \\ & | & \ldots & \text{as in section 4.1} \\ & | & e{:}a & \text{labelled term } (e \in E) \end{array}$$

There are new one-step reduction rules in addition to those of section 4.1:

$$\begin{array}{rcl} (e{:}b)(a) & \to & e{:}(b(a)) \\ (e{:}b)[s] & \to & e{:}(b[s]) \\ (e{:}b).l & \to & e{:}(b.l) \\ \text{if } (e{:}a) \text{ then } b \text{ else } c & \to & e{:}(\text{if } a \text{ then } b \text{ else } c) \end{array}$$

The grammar for active contexts is extended with one clause:

$$\begin{array}{rcll} C & ::== & & \text{active contexts} \\ & | & \ldots & \text{as in section 4.1} \\ & | & e{:}C & \text{labelled context } (e \in E) \end{array}$$

$$
\begin{array}{llll}
a, b, c & ::== & & \text{terms} \\
 & | & x & \text{variable } (x \in V) \\
 & | & \lambda x.a & \text{abstraction } (x \in V) \\
 & | & b(a) & \text{application} \\
 & | & a[s] & \text{closure} \\
 & | & \langle l_1 = a_1, \ldots, l_n = a_n, \mathit{else} = a_{n+1} \rangle & \text{record } (l_i \in L,\ \text{distinct}) \\
 & | & a.l & \text{selection } (l \in L) \\
 & | & \text{true} & \text{true} \\
 & | & \text{false} & \text{false} \\
 & | & \text{if } a \text{ then } b \text{ else } c & \text{conditional} \\
s & ::== & x_1 = a_1, \ldots, x_n = a_n, \mathit{else} = a_{n+1} & \text{substitutions} \\
 & & & (x_i \in V,\ \text{distinct})
\end{array}
$$

Figure 1. Grammar for the weak λ-calculus.

$$
\begin{array}{rcll}
x[x_1 = a_1, \ldots, x_n = a_n, \mathit{else} = a_{n+1}] & \rightarrow & a_i & (x = x_i) \\
x[x_1 = a_1, \ldots, x_n = a_n, \mathit{else} = a_{n+1}] & \rightarrow & a_{n+1} & (x \neq \text{all } x_i) \\
(b(a))[s] & \rightarrow & b[s](a[s]) & \\
((\lambda x.b)[s])a & \rightarrow & b[(x = a) \cdot s] & \\
(b.l)[s] & \rightarrow & (b[s]).l & \\
(\langle l_1 = a_1, \ldots, l_n = a_n, \mathit{else} = a_{n+1} \rangle)[s].l & \rightarrow & a_i[s] & (l = l_i) \\
(\langle l_1 = a_1, \ldots, l_n = a_n, \mathit{else} = a_{n+1} \rangle)[s].l & \rightarrow & a_{n+1}[s] & (l \neq \text{all } l_i) \\
\text{true}[s] & \rightarrow & \text{true} & \\
\text{false}[s] & \rightarrow & \text{false} & \\
(\text{if } a \text{ then } b \text{ else } c)[s] & \rightarrow & \text{if } a[s] \text{ then } b[s] \text{ else } c[s] & \\
\text{if true then } b \text{ else } c & \rightarrow & b & \\
\text{if false then } b \text{ else } c & \rightarrow & c &
\end{array}
$$

Figure 2. One-step reduction rules for the weak λ-calculus.

$$
\begin{array}{llll}
C & ::== & & \text{active contexts} \\
 & | & \_ & \text{hole} \\
 & | & C(a) & \text{application (left)} \\
 & | & b(C) & \text{application (right)} \\
 & | & a[S] & \text{closure} \\
 & | & C.l & \text{selection } (l \in L) \\
 & | & \text{if } C \text{ then } b \text{ else } c & \text{conditional (guard)} \\
 & | & \text{if } a \text{ then } C \text{ else } c & \text{conditional (then)} \\
 & | & \text{if } a \text{ then } b \text{ else } C & \text{conditional (else)} \\
S & ::== & x_1 = a_1, \ldots, x_i = C_i, \ldots x_n = a_n, \mathit{else} = a_{n+1} & \text{substitutions} \\
 & | & x_1 = a_1, \ldots, x_n = a_n, \mathit{else} = C & (x_i \in V,\ \text{distinct})
\end{array}
$$

Figure 3. Grammar for active contexts for the weak λ-calculus.

As usual, the congruence rule permits reduction in any active context:

$$\frac{a \to b}{C\{a\} \to C\{b\}}$$

for any active context C.

4.3 Dependency analysis and caching (by example)

The labelled calculus provides a basis for dependency analysis and caching. The sequence of definitions and results would be much as in section 3. We do not go through it, but rather give one instructive example.

We consider the term:

$$\begin{array}{l}((\lambda x.\, x.l_1)\langle l_1 = y_1, l_2 = y_2, \mathit{else} = w\rangle) \\ \qquad\qquad [y_1 = z_1, y_2 = z_2, \mathit{else} = w]\end{array}$$

This term yields z_1. We label the term, obtaining:

$$\begin{array}{l}((\lambda x.\, x.l_1)\langle l_1 = (e_1{:}y_1), l_2 = (e_2{:}y_2), \mathit{else} = (e_3{:}w)\rangle) \\ \qquad\qquad [y_1 = (e_4{:}z_1), y_2 = (e_5{:}z_2), \mathit{else} = (e_6{:}w)]\end{array}$$

This labelled term yields $e_1{:}e_4{:}z_1$, so we immediately conclude that the following prefix also yields z_1:

$$\begin{array}{l}((\lambda x.\, x.l_1)\langle l_1 = y_1, l_2 = \_, \mathit{else} = \_\rangle) \\ \qquad\qquad [y_1 = z_1, y_2 = \_, \mathit{else} = \_]\end{array}$$

Thanks to our definition of the prefix ordering, this prefix is equivalent to:

$$((\lambda x.\, x.l_1)\langle l_1 = y_1, \mathit{else} = \_\rangle)[y_1 = z_1, \mathit{else} = \_]$$

Suppose that, in our cache, we record this prefix with the associated result z_1; and suppose that later we are given the term:

$$\begin{array}{l}((\lambda x.\, x.l_1)\langle l_1 = y_1, l_3 = y_{17}(y_{17}), \mathit{else} = w'\rangle) \\ \qquad\qquad [y_{17} = z_1, y_1 = z_1, \mathit{else} = w']\end{array}$$

This term matches the prefix in the cache entry, so we immediately deduce that it reduces to z_1.

As this example illustrates, the labelled reductions help us identify irrelevant components of both substitutions and records. The prefix ordering and the use of *else* then allow us to delete those irrelevant components and to add new irrelevant components.

In some applications, irrelevant components may be common. For example, in the context of Vesta, a large record may bundle compiler switches, environment variables, etc.; for many computations, most of these components are irrelevant. In such situations, the ability to detect and to ignore irrelevant components is quite useful—it means more cache hits.

5 Conclusions

We have developed techniques for caching in higher-order functional languages. Our approach relies on using dependency information from previous executions in addition to the outputs of those executions. This dependency information is readily available and easy to exploit (once the proper tools are in place); it yields results that could be difficult to obtain completely statically. The techniques are based on a labelled λ-calculus and, despite their pragmatic simplicity, benefit from a substantial body of theory.

Acknowledgements

We thank Yanhong Liu and Tim Teitelbaum for discussions about related work; Allan Heydon, Jim Horning, Roy Levin, and Yuan Yu for discussions about Vesta; Alain Deutsch and Simon Peyton Jones for comments on strictness analysis; and Georges Gonthier for assorted remarks on a draft of this paper.

References

[ACCL91] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[Bar84] Henk P. Barendregt. *The Lambda Calculus.* North Holland, Revised edition, 1984.

[Ber78] G. Berry. Stable models of typed lambda-calculi. In *Proc. 5th Coll. on Automata, Languages and Programming*, Lectures Notes in Computer Science, pages 72–89. Springer-Verlag, 1978.

[BHA86] G. L. Burn, C. Hankin, and S. Abramsky. Strictness analysis for higher-order functions. *Science of Computer Programming*, 7:249–278, 1986.

[Fie90] John Field. On laziness and optimality in lambda interpreters: Tools for specification and analysis. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–15, 1990.

[FT90] John Field and Tim Teitelbaum. Incremental reduction in the lambda calculus. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 307–322. ACM, 1990.

[GLM92] Georges Gonthier, Jean-Jacques Lévy, and Paul-André Melliès. An abstract standardisation theorem. In *Seventh Annual IEEE Symposium on Logic in Computer Science*, 1992.

[GVS95] Milind Gandhe, G. Venkatesh, and Amitabha Sanyal. Labeled λ-calculus and a generalised notion of strictness. In *Asian Computing Science Conference*, Lecture Notes in Computer Science. Springer-Verlag, December 1995.

[HL91] Gérard Huet and Jean-Jacques Lévy. *Computations in Orthogonal Term Rewriting Systems.* MIT Press, 1991.

[HL93] Chris Hanna and Roy Levin. The Vesta language for configuration management. Research Report 107, Digital Equipment Corporation, Systems Research Center, June 1993. Available from http://www.research.digital.com/SRC.

[Hug85] John Hughes. Lazy memo-functions. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 129–146, September 1985.

[Klo80] Jan Willem Klop. *Combinatory Reduction Systems.* PhD thesis, CWI, 1980.

[Lév78] Jean-Jacques Lévy. *Réductions Correctes et Optimales dans le Lambda Calcul*. PhD thesis, University of Paris 7, 1978.

[LM93] Roy Levin and Paul R. McJones. The Vesta approach to precise configuration of large software systems. Research Report 105, Digital Equipment Corporation, Systems Research Center, June 1993. Available from http://www.research.digital.com/SRC.

[Mar91] Luc Maranget. Optimal derivations in weak lambda-calculi and in orthogonal term rewriting systems. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, 1991.

[Mic68] D. Michie. 'Memo' functions and machine learning. *Nature*, 218:19–22, 1968.

[O'D77] Michael O'Donnell. *Computing in Systems described by Equations*. PhD thesis, Cornell University, 1977.

[Pug88] William Pugh. *Incremental Computation and the Incremental Evaluation of Functional Programs*. PhD thesis, Cornell University, 1988.