چطور کاری کنیم لیسپ سریع‌تر از سی اجرا شود

دیدیر ورنا
didier@lrde.epita.fr

مترجم: فاروق کریمی زاده
fkz@riseup.net

۱۸ تیر ۱۳۹۹

فهرست مطالب

چکیده

برخلاف باور رایج، امروزه کد لیسپ میتواند بسیار کارآمد باشد: می‌تواند به اندازه سی سریع اجرا شود و حتی در مواقعی از سی نیز سریع‌تر اجرا شود. در این مقاله توضیح می‌دهیم چطور کد لیسپ را برای کارایی مناسب با معرفی تعریف‌های نوع مناسب، استفاده از ساختمان‌های داده مناسب و دادن اطلاعات به کامپایلر تنظیم کنیم. همچنین توضیح می‌دهیم چطور کامپایلرها این کارایی را بدست می‌آورند. این تکنیک‌ها برای الگوریتم‌های ساده‌ی پردازش تصویر استفاده می‌شوند تا کارایی ذکر شده برای دسترسی به پیکسل‌ها و اعمال جبری را در هر دو زبان نشان دهیم.
کلمات کلیدی: لیسپ،سی،پردازش تصویر،آنالیز عددی،کارایی

۱ معرفی

بیش از ۱۵ سال از فرایند استانداردسازی کامن لیسپ[۵] و بیش از ۱۰ سال بعد از اینکه مردم واقعا به کارایی اهمیت دادند[۱، ۴]، لیسپ هنوز از افسانه‌ی کند بودنش رنج می‌کشد. این ایده مدت‌ها پیش منسوخ شده است.امروزه لیسپ می‌تواند به اندازه سی یا حتی سریع‌تر از آن اجرا شود.
اگر این ایده نادرست کند بودن هنوز رایج است، احتمالا به دلیل نداشتن دانش یا اشتباه فهمیدن چهار فاکتور کلیدی برای دستیابی به کارایی در لیسپ است:

- **کامپایلرها** درحالی که لیسپ عموما به عنوان یک زبان مفسری شناخته می‌شود، کامپایلرهای بسیار خوب و سریعی وجود دارند که می‌توانند نه تنها بایت‌کد بلکه کد ماشین بومی را نیز از کد لیسپ تولید کنند.
- **بررسی گونه‌ای ایستا** درحالی که لیسپ بیشتر با بررسی گونه‌ای پویای خود شناخته می‌شود، اما استاندارد کامن لیسپ روش‌هایی برای تعریف نوع متغیرها به صورت ایستا(که در زمان کامپایل شناخته می‌شوند) ارائه می‌دهد.همانطور که در سی اینکار را می‌کنید.
- **سطوح ایمنی** با وجود اینکه کد لیسپ با بررسی گونه‌ای پویا باعث بررسی نوع پویا در زمان اجرا می‌شود اما این امکان وجود دارد که به کامپایلر بگوییم تمام بررسی‌های ایمنی را کنار بگذارد تا به کارایی مطلوب برسیم.
- **ساختمان‌های داده** اگرچه لیسپ با پردازش لیست شناخته می‌شود اما استاندارد کامن لیسپ نوع‌های داده‌ی بسیار کارا مانند آرایه‌های خاص، ساختمان‌ها یا جداول هش را ارائه می‌کند که لیست‌ها را تقریبا منسوخ می‌کند.

آزمایش‌ها روی کارایی لیسپ و سی در زمینه پردازش تصویر انجام شدند. ما چهار الگوریتم ساده را در هر دو زبان محک زدیم تا کارایی عمل‌های بنیادی و سطح پایین مانند دسترسی حجیم به پیکسل‌ها و پردازش جبری را بسنجیم. به عنوان یک نتیجه، کارایی یکسانی از دو زبان لیسپ و سی و حتی ٪۱۰ کارایی بیشتر از لیسپ در بعضی مواقع دیدیم.
در این مقاله الگوریتم‌های استفاده‌شده برای محک زدن و نتایج آزمایشی کارایی را که دریافت کردیم معرفی می‌کنیم و نمایش می‌دهیم. ما همچنین نشان می‌دهیم چطور کد متناظر لیسپ را با معرفی تعاریف مناسب نوع، استفاده از ساختمان‌های داده مناسب و دادن اطلاعات مناسب به کامپایلر تنظیم کنیم تا به کارایی ذکر شده دست پیدا کنیم.

۲ شرایط آزمایش

۱.۲ الگوریتم‌ها

آزمایش‌های ما بر پایه ۴ الگوریتم خیلی ساده بنا شده‌اند: انتساب پیکسل‌ها(ساخت یک تصویر با مقادیر ثابت) و اضافه،ضرب کردن و تقسیم کردن مقدار عددی پیکسل‌ها (پر کردن تصویر مقصد با جمع،ضرب و تقسیم مقدار عددی پیکسل‌ها با مقداری ثابت). این الگوریتم‌ها با وجود سادگی بسیار اما بسیار مرتبط هستند چراکه با عمل‌های بنیادی و سازنده پردازش تصویر درگیر می‌کنند: دسترسی به پیکسل و پردازش جبری.

رفتار الگوریتم‌ها با پارامترهای دیگر مانند اندازه تصویر، نحوه نمایش تصویر(خطی یا آرایه‌های چندبعدی) و غیره کنترل شده است. برای اختصار،تنها تعداد کمی از نتایج مرتبط اینجا به نمایش در می‌آیند.بااینحال اگر علاقه‌مند به دانستن دقیق ترکیب پارامترها و نتایج محکی که به دست آوردیم هستید،می‌توانید تمام کد منبع و نمودارهای مقایسه‌ای آزمایش را در وب‌سایت نویسنده[۱] بیابید. در ادامه این مقاله،تصاویر ۸۰۰ در ۸۰۰ را بصورت آرایه‌های یک بعدی از اعداد صحیح یا اعشاری نمایش می‌دهیم.خط‌ها به صورت متوالی و پشت سر هم هستند.

۲.۲ قرارداد

محک‌ها برای روی یک سیستم دبیان گنو/لینوکس[۲] که کرنل نسخه 2.4.27-2-686 اجرا می‌کند تولید شده‌اند. پردازنده نیز یک پنتیوم ۴ با فرکانس کاری ۳ گیگاهرتز با ۱ گیگابایت حافظه و ۱ مگابایت حافظه پنهان سطح ۲ می‌باشد. برای دوری از اثرات جانبی غیرقابل اندازه‌گیری سیستم‌عامل و سخت‌افزار تا جای ممکن، رایانه در حالت تک-کاربر بوت شده و کرنل بدون قابلیت چندپردازشی کامپایل شده بود(ویژگی فراریسمانی پردازنده غیرفعال شده بود)
همچنین جهت دوری از هر گونه اثر جانبی شروع برنامه، کارایی برای ۲۰۰ پیمایش هر الگوریتم اندازه گرفته شده است.

۳ برنامه‌های سی و محک‌ها

برای محک برنامه‌های نوشته شده به زبان سی از کامپایلر سی گنو[۳] نسخه 4.0.3(نسخه 4.0.3-1 دبیان) استفاده کردیم. با استفاده از O3- و DNDEBUG- و با inline[۴] کردن توابع در حلقه تکرار با ۲۰۰ تکرار بهینه‌سازی کامل انجام شده است. البته بهینه‌سازی بدست آمده از inline کردن تقریبا ناچیز است.قطعا قیمت صدا زدن توابع در مقایسه با زمانی که طول می‌کشد تا توابع اجرا شوند ناچیز است.
برای شفافیت یک برنامه نمونه(جزئیات حذف شده‌اند) در تکه کد شماره ۱ نمایش داده شده است.این تابع مقادیر ثابت را به تصاویر اعشاری اضافه می‌کند. جدول ۱ زمان اجرا شدن کدهای سی را برای اعداد صحیح و اعشاری برای تصویری ۸۰۰ در ۸۰۰ و با ۲۰۰ تکرار متوالی نشان می‌دهد. از این نتایج به عنوان یک مرجع برای کد لیسپ که در آینده به آن می‌پردازیم استفاده می‌کنیم.

تکه کد ۱: جمع پیکسل اعشاری،پیاده‌سازی در سی

```
void add(image *to, image *from, float val) {
    int i;
    const int n = ima->n;

    for (i = 0; i < n;++i)
        to->data[i] = from->data[i] + val;
}
```

خواننده ممکن است از کارایی تقسیم عدد صحیح که به عنوان یک عمل گران شناخته می‌شود تعجب کند. توضیح آن این است که زمانی که inline کردن فعال است کامپایلر گنو می‌تواند یک بهینه‌سازی به اسم «تقسیم عدد صحیح ثابت» را انجام دهد[۶] که به جای عمل تقسیم،ضرب و مقداری اضافه کردن انجام می‌دهد که نسبت به تقسیم صحیح ارزان‌تر هستند.

[۱] http://www.lrde.epita.fr/~didier/comp/research/
[۲] http://debian.org
[۳] http://gcc.gnu.org
[۴] برای این کلمه معادل فارسی مناسبی نیافتم در نتیجه از خود کلمه انگلیسی استفاده کردم.این کار در برنامه‌نویسی به این معنی است که کامپایلر به جای صدا زدن تابع،بدنه تابع را در کد وارد می‌کند.مثلا اگر تابع f را صدا بزنید با این روش کامپایلر به جای صدا زدن f از بدنه تابع استفاده می‌کند.مترجم

الگوریتم	تصویر صحیح	تصویر اعشاری
انتساب	0.29	0.29
جمع	0.48	0.47
ضرب	0.48	0.47
تقسیم	0.58	1.93

جدول ۱: مدت زمان اجرا به ثانیه،پیاده‌سازی در سی

۴ اولین تلاش با لیسپ

برای آزمایش کد لیسپ از شرایط آزمایشی که در بخش ۲ توضیح داده شد استفاده کردیم. همچنین چندین کامپایلر لیسپ را امتحان کردیم.برای اختصار، محک‌ها در ادامه این مقاله با CMU-CL[۵] نسخه ۱۹c CVS زده شده‌اند.
در این بخش اولین تلاش برای نوشتن کد لیسپ را که معادل تکه کد ۱ است توضیح می‌دهیم.این تلاش در تکه کد ۲ قابل مشاهده است.

تکه کد ۲ : جمع پیکسل،اولین نسخه در لیسپ

```
(defun add (to from val)
  (let ((size (array-dimension to 0)))
    (dotimes (i size)
      (setf (aref to i) (+ (aref from i) val)))))
```

استاندارد کامن لیسپ[۵] علاوه بر لیست‌ها که استفاده ازشان ناگریز است، نوع‌های داده‌ی مدرن‌تری نیز مانند ساختمان‌ها، آرایه‌ها و جداول هش ارائه می‌کند. آرایه‌ها به شما اجازه می‌دهند تا اشیای لیسپ را با یک سیستم مختصات خطی در آرایه ذخیره کنید یا از آرایه بخوانید. این را میتوان به مشابه malloc و calloc در سی در نظر گرفت.تابع لیسپ برای ساخت آرایه‌ها make-array است. در لیسپ آرایه‌ها میتوانند چند بعدی باشند.در تکه کد ۲ می‌توانید فراخوانی به array-dimension را ببینید که اندازه اولین ردیف آرایه را می‌گیرد(نیازی به ذخیره این مقدار در یک ساختمان داده مانند زبان سی نیست). بیاد بیاورید که از آرایه های تک بعدی خطی برای نمایش تصاویرمان استفاده می‌کنیم. تابع جهت دسترسی به عناصر آرایه aref نام دارد و انتساب به وسیله‌ی setf انجام می‌شود. dotimes یک ماکرو جهت ایجاد یک حلقه به روشی مانند حلقه for در زبان سی است.
اجرای این تابع در یک مفسر لیسپ نشان می‌دهد که این کد حدودا ۲۳۰۰ برابر کندتر از همین کد در زبان سی است(زمان سیستم و آشغال جمع‌کن در نظر گرفته نشدند). اما نسخه کامپایل شده تنها ۶۰ برابر کندتر از نسخه سی است. نهایتا حتی با فعال‌سازی بهینه‌سازی‌ها(بخش ۵.۵ را ببینید) هنوز کد لیسپ ۲۰ برابر کندتر از کد سی است.
برای اینکه بفهمیم چرا کد لیسپ کارایی پایینی دارد باید متوجه باشیم که در کد لیسپ‌ما برخلاف کد سی نوع‌ها مشخص نشده‌اند.متغیرها و آرگامون‌های(یا ورودی‌های) توابعی که استفاده می‌کنیم میتوانند هر شئ لیسپ را درون خود نگه دارند.مثلا ما از array-dimension برای to آرگامون تابع استفاده کردیم اما چیزی جلوی ما را نمی‌گیرد که چیزی جز آرایه وارد کنیم.ما از اعمال جبری برای آرگامون val استفاده می‌کنیم اما می‌توانیم چیزی جز عدد وارد کنیم.
به عنوان یک نتیجه کد لیسپ کامپایل شده باید به صورت پویا چک کند که با توجه به عمل‌گر هایی که استفاده می‌کنیم متغیرها نوع مناسبی دارند یا خیر. قدم بعدی ما باید ارائه اطلاعات مربوط به نوع‌ها به کامپایلر لیسپ باشد همانطور که اینکار را برای سی انجام می‌دهیم.

[۵] http://www.cons.org/cmucl

۵ تعیین نوع کد لیسپ

۱.۵ طرزکار تعیین نوع

استاندارد کامن لیسپ روش‌هایی برای تعریف نوع اشیای لیسپ زمان کامپایل ارائه میکند. البته هیچ‌کس مجبور نیست نوع متغیر را تعریف کند: نوع‌ها میتوانند زمانی که شناخته شدند تعریف شوند یا در غیر این صورت تعیین نشده باقی بمانند.از کامپایلرهای لیسپ انتظار می‌رود تمام تلاششان را با توجه به اطلاعاتی که دارند بکنند.
بعد از استاندارد سازی کامن لیسپ دیگر این گفته که لیسپ یک زبان با تعیین نوع پویاست صحیح نیست. لیسپ می‌تواند بسته به خواست برنامه‌نویس دارای تعیین نوع پویا یا ایستا باشد.
راه‌های مختلفی جهت مشخص نمودن نوع‌ها در کامن لیسپ وجود دارد.راه اول دادن آرگامون‌های بخصوص به توابع است. برای مثال اگر میخواهید یک آرایه ایجاد کنید و می‌دانید که این آرایه تنها اعداد اعشاری را درون خود نگه خواهد داشت میتوانید پارامتر کلیدی element-type: رابه تابع make-array بدهید مانند زیر:

```
(make-array size :element-type 'single-float)
```

راه دیگر مشخص نمودن نوع‌ها به وسیله‌ی «تعاریف» است: طرزکارش مشخص نمودن نوع یک آرگامون تابع یا یک متغیر کران‌دار است. یک تعریف نوع باید نزدیک اولین ظهور متغیری که به آن ارجاع می‌دهد قرار گیرد.تکه کد ۳ نسخه بعدی الگوریتم جمعمان را به همراه تعاریف نوع نشان می‌دهد.

تکه کد ۳ : جمع پیکسل‌ها،نسخه تعیین نوع شده لیسپ

```
(defun add (to from val)
  (declare (type (simple-array single-float (*))
                 to from))
  (declare (type single-float val))
  (let ((size (array-dimension to 0)))
    (dotimes (i size)
      (setf (aref to i) (+ (aref from i) val)))))
```

همانطور که می‌بینید،ما نوع (مورد انتظار) متغیرهای سه پارامتر تابع را مشخص کردیم: دو آرایه با مقادیر single-float (در کامن لیسپ simple-array به معادل آرایه‌های سی هستند.انواع دیگری نیز وجود دارند که در اینجا به آن نمی‌پردازیم.)، و یک پارامتر از نوع single-float همانطور که می‌بینید. (*) یا یک آستریک در پرانتز به این معنی است که آرایه‌ها تک بعدی هستند. آرایه‌های دو بعدی با (* *) نشان داده می‌شوند.
روش سومی برای ارائه تعاریف نوع نیز وجود دارد که در بخش ۲.۶ توضیح داده می‌شود.

۲.۵ نمایش اشیا

برای اینکه بفهمیم چرا تعاریف نوع برای بهینه‌سازی کارایی لیسپ مهم هستند بایست مفاهیم تعریف نوع پویا را کمی درک کنیم.از آنجا که اشیای لیسپ می‌توانند از هر نوع باشند(یا حتی بدتر!از هر اندازه)، متغیرها در لیسپ با خود اطلاعات راجب به نوع را حمل نمی‌کنند. بلکه اشیا خودشان مسئول ارائه نوع خودشان هستند.این به این معنیست که اشیای لیسپ بیشتر اوقات حامل اطلاعات راجب نوعشان به همراه یک اشاره‌گر به مقدار واقعیشان هستند.به روشنی ارجاع اشاره‌گرها باعث از دست رفتن مقدار زیادی از کارایی می‌شود.
زمانی که اطلاعات نوع در دسترس است چندین تکنیک برای عدم استفاده از روش نمایش اشاره‌گرها وجود دارد.[۱] دو روش از این روش‌ها توضیح داده شده است.

۳.۵ طرح‌بندی انباره آرایه

اولین مثال با طرز ذخیره اشیای لیسپ در آرایه‌ها سرکار دارد: اگر کامپایلر لیسپ می‌داند که برای مثال یک آرایه تنها مقادیر اعشاری را نگه می‌دارد،می‌تواند مقادیر را مستقیما،در قالب بومی ماشین،در آرایه نگه دارد(درست مانند سی) به جای اینکه اشاره‌گرهایی به آن‌ها نگه دارد. یک نسخه خاص از تابع aref که جهت دسترسی مستقیم به مقادیر به جای ارجاع به اشاره‌گرها استفاده می‌شود.این نسخه حتی می‌تواند inline شود. فرآیند خاص سازی توابع به جای استفاده از نسخه‌های عمومی توابع در جامعه لیسپ «کدزنی باز» نامیده می‌شود.

۴.۵ اشیای بلافاصله

مثال دوم ما که یک شئ بلافاصله است،شیئی است که به اندازه کافی کوچک است که در یک کلمه از حافظه ذخیره شود و نیازی به ارجاع به اشاره‌گر نباشد. در پیاده‌سازی های مدرن لیسپ تمامی مقادیر اشاره‌گرهایی معتبر به اشیای لیسپ نیستند: معمولا ۳ بیت کم ارزش جهت ذخیره اطلاعات نوع رزرو شده‌اند. در کامن لیسپ نوع عدد صحیح استاندارد fixnum است. کامپایلر CMU-CL [۳] این نوع عدد صحیح را در کلمات حافظه نگه می‌دارد و دو بیت پایانی را صفر قرار می‌دهد. این روش روی یک ماشین ۳۲ بیتی دقت ۳۰ بیتی می‌دهد.اکثر عملگرها بر روی fixnum ها به صورت مستقیم اعمال می‌شود و عده کمی از آنها نیازمند شیفت بیتی هستند که به هر حال خرج کوچکیست.
اکنون،اجازه دهید کد اسمبلی تولید شده توسط CMU-CL را برای یک حلقه‌ی dotimes ببینیم.این کد در تکه کد ۴ نمایش داده شده است.
خطوط جالب با L۰ و L۱ مشخص شده‌اند.این خطوط مربوط به افزایش شاخص و مقایسه با حد بالا که ۱۰۰ است هستند.اینجا متوجه می‌شویم که کامپایلر مقادیر را چنان وفق داده تا مستقیم با نمایش شیفت داده شده عدد صحیح کار کند که به اندازه اعداد صحیح خود ماشین سریع هستند.
می‌بینیم که کامپایلرهای لیسپ تلاش می‌کنند هوشمند باشند و بایست همینطور باشد و هر نوع بهینه‌سازی را پیاده سازی کنند.

تکه کد ۴ : دیس‌اسمبلی یک حلقه‌ی dotimes

```
90:       POP     DWORD PTR [EBP-8]
93:       LEA     ESP, [EBP-32]
96:       XOR     EAX, EAX
98:       JMP     L1
9A: L0:   ADD     EAX, 4
9D: L1:   CMP     EAX, 400
A2:       JL      L0
A4:       MOV     EDX, #x2800000B
A9:       MOV     ECX, [EBP-8]
AC:       MOV     EAX, [EBP-4]
AF:       ADD     ECX, 2
B2:       MOV     ESP, EBP
B4:       MOV     EBP, EAX
B6:       JMP     ECX
```

۵.۵ بهینه‌سازی

برای اینکه از تعیین نوع اشیای لیسپ چیزی بدست بیاوریم باید چیزی در مورد بهینه‌سازی‌ها بدانیم: استاندارد کامن لیسپ «کیفیت‌هایی» تعریف می‌کند که ممکن است کاربر علاقه‌مند به بهینه‌سازی آن‌ها باشد. سطح بهینه‌سازی معمولا توسط عددی صحیح از ۰ تا ۳ مشخص می‌شود.۰ به این معنیست که کیفیت مورد نظر اصلا مهم نیست و ۳ به این معناست که کیفیت به شدت مهم است. دو کیفیت از این کیفیت‌ها safety(چک کردن خطای زمان اجرا) و speed(سرعت کد) هستند.
توجه کنید که کیفیت بهینه‌سازی‌ها میتواند با اعلامیه‌ها به صورت جهانی مشخص شوند. همچنین می‌توانند برای هر تابع با تعاریف،مانند تعیین نوعی که پیشتر استفاده کردیم،مشخص شوند. برای تولید یک کد کاملا امن باید این طور تعریف کنید:

```
(declaim (optimize (speed 0)
                   (compilation-speed 0)
                   (safety 3)
                   (debug 3)))
```

زمانی که از کامپایلرهای لیسپ کد امن درخواست می‌شود،یک کامپایلر مدرن لیسپ با تمام تعاریف نوع به مشابه یک ادعا برخورد می‌کند و زمانی که مقادیر از نوع مورد انتظار نیستند، خطا می‌دهد و این فرآیند چک کردن زمان اجرا زمان می‌برد. زمانی که درخواست می‌شود کد سریع و ناامن تولید گردد، کامپایلر به تعاریف

	تصویر صحیح		تصویر اعشاری	
الگوریتم	سی	لیسپ	سی	لیسپ
انتساب	0.29	0.29	0.29	0.29
جمع	0.48	0.48	0.47	0.46
ضرب	0.48	0.48	0.46	0.45
تقسیم	0.58	1.80	1.93	1.72

جدول ۲: مدت زمان اجرا به ثانیه،پیاده‌سازی در سی و لیسپ

نوع «اعتماد» می‌کند و انواع بهینه‌سازی مانند عدم استفاده از نمایش اشاره‌گری برای اشیای لیسپ هر زمان که ممکن باشد،کدزنی باز توابع و غیره را اعمال می‌کند. به عنوان یک نتیجه، درست مثل سی اگر مقادیر از نوع‌های مورد انتظار نباشند،رفتار کد تعریف نشده است.
آزمایش‌های ما نشان می‌دهد که کد لیسپ کاملا امن از معادل آن در سی ۳۰ برابر کندتر اجرا می‌گردد.از منظری دیگر، جدول ۲ نتایج را برای کد کاملا بهینه‌سازی شده(ناامن) نمایش می‌دهد.برای مقایسه نتایج مربوط به کد سی نیز نوشته شده است.

در مورد تصاویر صحیح،میبینیم که سی و لیسپ دقیقا سرعت یکسانی دارند،جدای از اینکه تقسیم در لیسپ ۳ برابر کندتر است. بعد از تحقیق بیشتر و دیس‌اسمبلی کد دودویی تولید شده،بنظر می‌آید که هیچ‌کدام از کامپایلرهای لیسپ آزمایش شده از بهینه‌سازی تقسیم عدد صحیح ثابت که کامپایلر گنو می‌تواند انجام دهد آگاه نیستند، در نتیجه از دستور کند idiv متعلق به خانواده پردازنده‌ی x86 استفاده می‌کنند. این باعث تاسف است اما بنظر نمی‌آید بهبود کامپایلرها برای این مورد خاص زیاد سخت باشد.
اما در مورد تصاویر اعشاری کمی باعث تحیر است:می‌بینیم که ۳ الگوریتم اول،لیسپ در مدت زمانی به اندازه کد سی یا اندکی بهتر از آن اجرا می‌شود اما الگوریتم تقسیم در لیسپ ٪۱۰ سریع‌تر از سی است. همچنین در مواقعی دیگر که اینجا نمایش داده نشده است مشاهده شده که لیسپ مقدار قابل توجهی سریع‌تر است. این نتایج بایست به ما کمک کند تا توجه مردمان سی را جلب کنیم.

۶ استنتاج نوع

طرزکاری که تعیین نوع به کامپایلر کمک می‌کند بهینه‌سازی انجام دهد آن چنان که به نظر می‌آید ساده نیست. برای مثال توجه کنید که در تکه کد ۳ تمام متغیرها صریحا تعیین نوع نشده بودند. در واقع دو مشکل تا به الآن نادیده گرفته شدند:هیچ اطلاعاتی راجب اندازه آرایه ارائه نشده است پس ظرفیت عدد صحیح لازم برای متغیرهای size و i نامعلوم است. دوم اینکه نتیجه یک عمل جبری ممکن است با نوع عملوندهایش یکی نباشد.
زمانی که تمام نوع‌ها توسط برنامه‌نویس ارائه نشده‌اند کامپایلرهای مدرن لیسپ تلاش می‌کنند تا نوع‌های نامعلوم را از اطلاعات در دسترس «استنتاج» کنند. متاسفانه استاندارد آزادی زیادی در مورد اینکه چگونه با تعاریف نوع رفتار شود در اختیار کامپایلرها قرار می‌دهد و سامانه‌ی استنتاج نوع ممکن است در رفتار و کیفیت در کامپایلرهای مختلف متفاوت باشد. به عنوان یک مثال دو مشکل بالقوه با تعیین نوع توضیح داده شده‌اند.

۱.۶ حساب حلقه

بدون ارائه تعریف نوع CMU-CL میتواند اعمال جبری را بر روی شاخص حلقه‌ی ماکروی inline dotimes کند که توضیح می‌دهد چرا تعریف نوع صریحی برای i در تکه کد ۳ ارائه نکردیم. زمانی که دو حلقه‌ی dotimes به صورت تودرتو داشته باشیم، CMU-CL درخواست یک تعیین نوع صریح برای شاخص اولی می‌کند. درحالی که هیچ تعریفی برای دومی درخواست نمی‌کند.دلیل این رفتار در حال حاضر نامشخص است(حتی برای نگه‌دارندگان CMU-CL زمانی که با آن‌ها تماس گرفتیم) در این مورد مشکل است بفهمیم باگ در سامانه‌ی استنتاج نوع است یا در مکانیسم یادداشت‌های کامپایل. تنها را برای اینکه مطمئن شویم تمام بهینه‌سازی‌های ممکن اعمال شده است، دیس‌اسمبلی کد مشکوک است.

۲.۶ نتایج جبری

از سوی دیگر سامانه‌ی استنتاج نوع CMU-CL مزایای خودش را دارد.الگوریتم ضرب عدد صحیح که بر روی تصاویر صحیح اعمال می‌گردد را در تکه کد ۵ فرض کنید.

تکه کد ۵ : ضرب پیکسل،نسخه تعیین نوع شده در لیسپ

```
(defun mult (to from val)
  (declare (type (simple-array fixnum (*))
                 to from))
  (declare (type fixnum val))
  (let ((size (array-dimension to 0)))
    (dotimes (i size)
      (setf (aref to i) (* (aref from i) val)))))
```

باید یادآوری کنم که نتیجه ضرب دو fixnum ممکن است از یک fixnum بزرگ‌تر باشد(در واقع یک bignum). با این حال سامانه تعیین نوع CMU-CL متوجه می‌شود که ما نتیجه را در یک آرایه‌ی fixnum نگه می‌داریم. به عنوان یک نتیجه کامپایلر فرض می‌کند که انتظار داریم نتیجه یک fixnum باقی می‌ماند و به استفاده از حساب بهینه‌سازی شده ادامه می‌دهد.
متاسفانه تمام سامانه‌های استنتاج نوع به این هوشمندی نیستند.برای مثال با همین کد کامپایلر الگرو(ACL)[۶] در عوض سرعت،از یک تابع ضرب عمومی استفاده می‌کند که این توانایی را دارد تا در صورت نیاز bignum بربگرداند. از آنجا که می‌دانیم نتیجه ضرب یک fixnum باقی می‌ماند باید توسط یک مکانیسم تعیین نوع دیگر صریحا به ACL بگوییم که نتیجه fixnum باقی می‌ماند.همانطور که در تکه کد ۶ می‌بینید.

تکه کد ۶ : ضرب پیکسل،نسخه دوم در لیسپ

```
(defun mult (to from val)
  (declare (type (simple-array fixnum (*))
                 to from))
  (declare (type fixnum val))
  (let ((size (array-dimension to 0)))
    (dotimes (i size)
      (setf (aref to i)
            (the fixnum (* (aref from i) val))))))
```

این باعث تاسف است چرا که برنامه نویسان را مجبور می‌کند تا با تعاریفی که به صورت ایده‌آل نیازی نیست تنها برای قابل حمل بودن کد،کد را بهم بریزند.

۷ نتیجه

در این مقاله توضیح دادیم چطور در لیسپ با استفاده از ساختمان‌های مناسب داده،تعاریف نوع و تنظیمات بهینه‌سازی به کارایی برسیم.
از آنجا که کارهای زیادی بر روی کامپایلرها لیسپ انجام شده است[۱، ۴] تا به این نقطه برسیم که کد های معادل لیسپ و سی اکیدا کارایی یکسان یا حتی بعضی مواقع کارایی بهتری را در لیسپ شاهد هستیم. یادآوری کنم که زمانی که از کد یکسان یا معادل سی و لیسپ صحبت می‌کنیم،این گفته نسبتا نادقیق است.برای مثال داریم یک ساختار زبان(for) را با یک ماکروی برنامه‌نویس(dotimes) مقایسه می‌کنیم.همچنین داریم صدازدن توابع در سی را با توابعی در لیسپ مقایسه می‌کنیم که ممکن است به صورت پویا بازتعریف شوند. این به این معنی است که با روشنی ذاتی لیسپ،کامپایلرها باید حتی هوشمندتر باشند تا به کارایی سی برسند و این واقعا خبر خوبی برای جامعه‌ی لیسپ است.
البته لیسپ هنوز نقطه ضعف‌هایی دارد.دیدیم که تعیین نوع کد لیسپ بصورت درست و تا حد امکان بصورت ناچیز کاملا روشن نیست(بدون بهم ریختن کد). کامپایلرها ممکن است در مواجه با تعاریف نوع بسیار متفاوت رفتار کنند و ممکن است سامانه‌های استنتاج نوع متفاوتی با کیفیت‌های متفاوت ارائه دهند. شاید استاندارد کامن لیسپ زیادی آزادی در این زمینه برای کامپایلرها قرار می‌دهد.

[۶] http://franz.com

۸ چشم‌اندازها

با زاویه دیدی سطح پایین،توسعه محک‌هایمان برای کامپایلرها و معماری‌های مختلف و همچنین وارد شدن به گزینه‌های بهینه‌سازی هر کامپایلر برای دستیابی به کارایی بیشتر،می‌تواند جالب باشد.
از دیدی سطح متوسط،محک زدن الگوریتم‌های بسیار ساده برای ایزوله کردن پارامترهایی که میخواستیم تست کنیم(دسترسی به پیکسل و اعمال جبری)،ضروری بود. همین آزمایش‌ها بایست برای الگوریتم‌های پیچیده‌تر اختیار شود تا اثر آن را بر روی کارایی بفهمیم.
معمولا می‌تواند جالب باشد که شی‌گرایی پویا را با سی‌پلاس‌پلاس و CLOS مقایسه کنیم.[۲] در قدم بعدی،باید قابلیت‌های فرا-برنامه‌نویسی سامانه قالب‌بندی سی‌پلاس‌پلاس با توانایی لیسپ در مورد کامپایل توابع جدید در پرواز[۷] مقایسه شوند.
نهایتا هرکس باید متوجه باشد که نتایج کارایی که بدست آوردیم محدود به حوزه‌ی پردازش تصویر نیستند. هر برنامه که با محاسبات عددی بر روی مجموعه بزرگی از داده سرکار دارد ممکن است علاقه‌مند به دانستن کارایی‌ای که لیسپ ارائه می‌دهد باشد.

مراجع

[۱] Richard J. Fateman, Kevin A. Broughan, Diane K. Willcock, and Duane Rettig. Fast floating-point processing in Common-Lisp. ACM Transactions on Mathematical Software. 21(1):26-62, March 1995. http://openmap.bbn.com/~kanderso/performance/postscript/lispfloat.ps

[۲] Sonja E. Keene. Object-Oriented Programming in Common-Lisp: a Programmer's Guide to CLOS. Addison-Wesley, 1989.

[۳] Robert A. Mac Lachlan. The Python compiler for CMU-CL. In ACM Conference on Lisp and Functional Programming, pages 235-246. http://www-2.cs.cmu.edu/~ram/pub/lfp.ps

[۴] J.K. Reid. Remark on "fast floating-point processing in Common-Lisp". In ACM Transactions on Mathematical software, volume 22, pages 496-497. ACM Press, December 1996.

[۵] Guy L. Steele. Common-Lisp the Language, 2nd edition. Digital Press, 1990. http://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html

[۶] Henry S. Warren. Hacker's Delight. Addison Wesley Professional, July 2002. http://www.hackersdelight.org

[۷] on the fly